UCRL-ID-137634

TIGER LDRD Final Report

D. J. Steich, S. T. Brugger, J. S. Kallman, D. A. White

February 1, 2000

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

TIGER LDRD Final Report

EXECUTIVE SUMMARY

This final report describes our efforts on the Three-Dimensional Massively Parallel CEM Technologies LDRD project (97-ERD-009). Significant need exists for more advanced time domain computational electromagnetics modeling. Bookkeeping details and modifying inflexible software constitute a vast majority of the effort required to address such needs. The required effort escalates rapidly as problem complexity increases. For example, hybrid meshes requiring hybrid numerics on massively parallel platforms (MPPs).

This project attempts to alleviate the above limitations by investigating flexible abstractions for these numerical algorithms on MPPs using object-oriented methods, providing a programming environment insulating physics from bookkeeping.

The three major design iterations during the project, known as TIGER-I to TIGER-III, are discussed. Each version of TIGER is briefly discussed along with lessons learned during the development and implementation. An Application Programming Interface (API) of the object-oriented interface for Tiger-III is included in three appendices. The three appendices contain the Utilities, Entity-Attribute, and Mesh libraries developed during the project. The ~~three~~-API libraries represent a snapshot of our latest attempt at insulated the physics from the bookkeeping.

INTRODUCTION

Numerical simulation of PDEs is one of the cornerstones of modern engineering and physics. PDEs describe an enormous variety of phenomena: electromagnetic radiation, structural dynamics, thermal and fluid flow are just a few of the subject areas amenable to numerical simulation. Over the years, a great number of computer programs have been written to perform this simulation, and almost all of them have a great deal in common. They all must represent the domain on which the problem is to be solved, they all must use domain information to generate solver matrix coefficients, and if they deal with large enough problems, they must partition and distribute the problem over multiple processors and handle communications between partitions. Considering all of the codes that have been written and are being written, there is a huge duplication of effort.

The purpose of the TIGER project is to investigate abstractions that would make coding easier for the authors of numerical PDE simulation systems by extracting the bookkeeping and separating it from the physics. Examples of bookkeeping include representing (and extracting information from) problem meshes, as well as partitioning and communications. The vehicle for this effort was the C++ language. The object-oriented features of C++ make abstracting the bookkeeping into a small set of classes much less difficult.

To this end a set of libraries and a pre-mesh-processor has been constructed:

- A Utilities library.
  - set of container classes suited to PDE (Partial Differential Equation) solution
  - set of communication classes (Message Passing Interface (MPI) & Pthreads)
  - a memory management system that organizes raw memory usage
  - miscellaneous utilities (Oct-tree, Red-black binary tree, Registry, String, .. classes)
- A Entity-Attribute library.
  - Attribute class
    - Automates the organization and construction of local, shared, and computed data types in a multi-processor settings
  - Entity and Mesh_entity classes
    - Generic container~~s which~~ abstract ng data type ~~and~~ manage s the modification of Attributes
  - Species class
    - Manages collections of Entities having an identical sets of Attributes
  - extensive data manipulation methods → manages the modification of Entities
  - extensive data construction methods
  - multi-processor Entity/Attribute communication systems
  - extensible programming API for data modification and association
- A Mesh library.
  - ~~Set of~~ Mesh classes that allow ability to have multiple unconnected/ uncoordinated /unstructured / structured meshes
  - manages Topological connectivity between Entities
  - manages the construction of mesh parts into the Entity/Attribute container
  - serial and parallel versions Mesh reader hierarchy
- PreTiger a pre-mesh-processor- that sets up the mesh input for TIGER in multi-processor setting
  - manages the mesh part partitioning for structured /unstructured mesh parts
  - re-maps non-contiguous to contiguous ~~node/cell~~ numbers
  - allocates processors for each mesh part
  - ~~calls structured or unstructured partitioner based on mesh-type~~
  - writes ~~out~~ processor specific partition files for TIGER

These libraries allow the physicist or analyst to deal with the phenomena to be modeled instead of writing code to hold and distribute meshes in memory.

try

This project has gone through a number of design and implementation iterations. We have taken earlier designs to the point of building specialized Maxwell's equation simulators. At each iteration we have improved the performance and the programmer interface to the libraries. The first iteration culminated in TIGER-I, which was capable of representing unstructured meshes. The second iteration culminated in TIGER-II which was capable of representing hybrid (structured & unstructured) meshes with limited Attribute tagging and had very limited parallel communication capabilities. The result of the final iteration is TIGER-III, which can represent and communicate arbitrarily tagged hybrid meshes. Coupled with the physicist/analyst's code and a parallel solver ~~(such as PETSc),~~ it is hoped TIGER-III takes much of the pain out of writing PDE simulation suites. At this time, the TIGER-III implementation is in the final stages of completion.

Background

The primary purpose of this project was to investigate the use of object-oriented techniques to abstract physics from bookkeeping. The motivation behind accomplishing this task was to reduce the coding development time of complex applications. and unify the coding redundancy between various codes into a single platform Our primary focus was to have this applicable to time domain Computational ElectroMagnetics (CEM) software development with the intention of having the techniques developed here be useful to a much larger computational arena. The project software is named TIGER (TIme Domain Generalized Excitation and Response).

The two largest existing time domain codes CEM within engineering, TSAR (Time domain Source And Response), a structured Finite Difference Time Domain (FDTD) code, and DSI (a Discrete Surface Integral (DSI), an unstructured Finite Volume Time Domain (FVTD) code both have severe extensibility limitations. There are numerous versions of both these codes each with different feature sets. So in addition having code that was difficult to enhance, when adding new physics to the code, there was a dilemma as to which version of the code to add it to. Often the solution was to add it to the version closest to the problem of interest with the intention of going back later and adding the new physics to other versions in the future. Needless to say the solution had many difficulties. As the complexity of the problem increased the limitations mentioned above only increased.

The question was how

~~Our intent was to enhance the existing physics programming environment so that it significantly eased the programming burden of adding new physics to the code. At the same time we wanted to unify the redundancy between the various codes into a single platform, (i.e., programming environment), that could be used for future physics extensions.~~

In a typical time domain CEM code, over 90 percent of the code is devoted to bookkeeping issues. Even the source code within the physics kernel routines themselves is often dominated by bookkeeping. For example, a typical version of TSAR is almost entirely bookkeeping software. Even the core FDTD, boundary condition, source, and sensor routines are still primarily involved with bookkeeping issues. This is also the case for the DSI code, although there are more equations there are also more bookkeeping issues.

Approach

This section describes what we are trying to accomplish and an overview of our approach.

In order to alleviate the difficulties mentioned in the background section, we originally proposed the following four primary objectives:

"

1) to design an object-oriented framework that provides long term extensibility
2) to investigate abstractions required to efficiently insulate physics from bookkeeping
3) to develop generalized algorithms that allow the unification of existing time domain CEM technologies
4) to formulate and validate the new physics technologies required for Advanced Hydro Facility based on objectives 1-3.

"

During the course of the research, we have ~~changed~~ refined our emphasis and details of the approach. We briefly describe the reasons for these changes in the TIGER design sections of this report.

In simple terms, given a new, potentially unforeseen computational programmatic need, we desire to minimize the man-effort required to solve the problem. The most obvious answer is to already have the software required to solve the problem and have that software easy to use. If the need requires a new software technology then the effort required should be minimized, in part, by maximizing the reuse of existing software.

From a programmer's point of view, our primary objective is to investigate the use of object-oriented techniques to minimize the effort involved writing a time domain CEM software application code. There are several aspects to this objective. First, we would like to minimize the effort involved in building a new code from start to finish, ~~Second, we would like any new efforts to reuse as much of the existing software as possible so that a vast majority of the time we just need to incrementally build on the existing software. Thirdly, we would like to unify and generalize as much of the software development process in order to prevent or greatly circumvent the need to even build new software modules. Fourth, we would like the software built to be "designed" as flexible. That is, written with the attitude that it "will be extended" and not so much it "might be"~~ extended. Fifth, ~~we would like~~ the technologies developed here to be as applicable as possible to as many other Engineering and LLNL codes as possible.

facilitate the reuse of existing software, unify and generalize as much of the software development process as possible, build flexible software written with the attitude it will be extended, and lastly to have

most of the time

as possible

There is much overlap between each of the ~~five~~ aspects listed above. ~~The first aspect would reduce overall development time~~. If ~~we could build~~ it were possible to build an application code from start to finish measured in days/weeks rather than months/years then the need for flexibility, extensibility ~~is greatly~~ would be greatly diminished. because a ~~A programmer~~ programmer could ~~just~~ build a new code just as fast as he or she could modify an existing code.

While possible for simpler, more academic problems, world class, grand-challenge, like codes can not be built from start to finish in days at this point in time. So while reducing overall start-to-finish code-development cycle is important ~~we also have the other four~~ the other aspects of the objective ~~that help reduce the coding effort~~ are equally important.

Our approach to achieving the desired objectives was to build a set of object-oriented libraries that provide capabilities which reduce code development time. The intent behind these libraries was to have them be highly extensible and flexible and have the library interfaces themselves serve as the generalization and code-reuse vehicles.

The libraries would be an investigation into the automation of the bookkeeping involved in building a time domain CEM code. Our goal was to accomplish this with minimal loss of efficiency. We also desired this technology to be applicable to other computational areas.

Design Overview of TIGER-I applied

We began by investigating how to insulate the physics from the bookkeeping in areas where the majority of the software development time has occurred in the past. For time domain CEM codes, a vast majority of the software development effort has been in application of physics algorithms to mesh components. The bookkeeping involved in building, organizing, traversal, and querying of data structures constitutes a majority of the source code. Was there a way to unify this bookkeeping once and for all? Was it possible to pull the data details away from the physics? Was it possible to unify the hybrid nature of the mesh types, especially structured/unstructured differences? The following was our first attempt.

The mesh management system for TIGER-I consisted of an eight layered wedding cake of code, looking much like this:

```
             Mesh
     Unstructured_manager
        Submesh_manager
         Table_manager
           Superset
             Table
         Array_of_sets
   Memory and Array Classes
```

We will describe this wedding cake from the bottom up.

At the time this project started, the C++ language was still having features and libraries added. The Standard Template Library (STL) was neither fast nor memory efficient. But perhaps the biggest limitation was our strong desire to control the behavior of the data placed in these STL containers. Having the data remain fixed during certain construction phases and then relocate during others etc… are tasks the STL was not very suited for. It was decided that we would build our own memory manipulation and efficient array classes. The memory manipulation routines allowed us to keep track of memory (and were of great utility in debugging). The array classes were what everything else was build upon. The most important array classes were those that could change size: Adjustable_arrays and Noncontiguous_arrays. Adjustable arrays are arrays that can change their size dynamically. When a user accesses an element beyond the current array size, the array expands to include the newly accessed portion of the array. Adjustable arrays can be expensive, because if the array is already large and the user asks it to expand, it is possible that the entire array may have to be copied (if the array is in a part of memory where it can't easily expand and must be moved outright). Non-contiguous arrays are accessed in the same way that an Adjustable array is, but, behind the scenes, hidden from the casual programmer's eye, memory is allocated in blocks, and expansion never causes the wholesale moving of all of the data in the array. For large arrays, Non-contiguous storage is very efficient.

Above the array classes is the Array_of_sets. Consider a space filling mesh. The mesh has 0-dimensional nodes, 1-dimensional edges, 2-dimensional faces, and 3-dimensional cells. A node has a position, as well as a set of edges that are connected to it. An edge has two nodes that define its endpoints, as well as a set of faces that it helps define. A face has a set of edges, and, unless it is on the mesh surface, serves to separate two cells. A cell has a set of faces that define it. At the time we were writing TIGER-I, it would have been possible to make an individual C++ object out of each node, edge, face, and cell, but it would have been very expensive in terms of memory. As a way to consolidate some of these objects into more efficient structures we built Arrays-of-sets. The sets refer to the collections of the things that these objects point to (either directly or indirectly). An Array_of_sets was a specialization of a Noncontiguous_array that held a set of a particular size. Sets could be accessed, inserted, and removed.

Before describing the Table, we have to make a small detour and describe the entire structure necessary to hold an unstructured mesh. What is necessary is a set of four data structures, one of which represents cells, faces, edges, and nodes. The structure can be thought of as a pillar or column. At the top of the column is an array that, given an Entity number, will point to the sets that tell what that Entity is connected to (the ups (entities of higher dimension), and downs (entities of lower dimension)), and what the Entity's Attributes are. If you have a face number, you can get the face's sets, which will tell you the numbers of the cells the face is separating, as well as the numbers of the edges that make up the face. In addition you can get the face's Attribute (an integer tag). Similar arrangements hold for cells, edges, and nodes, although for nodes there is an additional array that is needed to hold the node positions.

Tables contain Arrays_of_sets for the ups, downs, and Attributes of a group of entities, all of which have the same number of ups and downs. In any of the columns mentioned in the previous paragraph there will be a number of tables. For instance, in the edge column, there will be tables for edges with two faces and two nodes, three faces and two nodes, four faces and two nodes, etc.

A Super_set is used as an accessor in a Table_manager. It contains member functions that let the programmer access the ups, downs, and Attributes of any Entity. The

Super_set is also the way a programmer adds ups, downs, and Attributes. When the user asks the Table_manager for information about an Entity, it gets passed back in a Super_set.

The Table_manager is, for the most part, the column mentioned previously. It manages all of the tables necessary to describe, for instance, all of the edges in a mesh. If an Entity has an up or a down added to or removed from it, the table manager takes it from its current table and puts it into the appropriate one. This happens without the knowledge or interference of the programmer.

The Submesh_manager is an abstract base class that represents all things that manage the four columns mentioned earlier. It has placeholders for methods for accessing Table_managers for each of cells, faces, edges, and nodes as well as loading of all of these entities. The reason this class is abstract is that it was supposed to be a unified interface to both structured and unstructured sub-meshes.

The Unstructured_manager implements the Submesh_manager interface for unstructured meshes. In addition to the cell, face, edge, and node Table_managers, it also contains an array to hold global node numbers (important in the construction of the mesh). It has the actual methods for accessing the Table_managers and loading the mesh.

The Mesh class is meant to hold a set of Submesh_managers and deal with their interfaces. Mesh currently has the ability to hold a single Unstructured_manager. The Mesh class was used in a prototype physics code that was used to model electron accelerator components.

At this point we ~~has~~ d constructed a prototype Mesh management system. Next, we tested the prototype version out to see how successfully the bookkeeping was abstracted from the physics. We built a finite volume DSI formulation time domain electromagnetics code from scratch using our Mesh management system. The original code ~~was~~ developed by Neil Madsen at LLNL, was a FORTRAN based code that took many man-years to develop. To build and debug that code from scratch without using Mesh management several system would have a tremendous effort probably requiring at least a ~~man-year. (In~~ fact, another FORTRAN version had ~~been built that had~~ required man-years but included particles). Our Mesh management system was far from complete. The software required iterators, better data manipulation support, and much better interfaces but it was far enough to test.

A great deal of effort went into this test. We successfully added the DSI physics kernel and complex sources and sensors. In addition, modifications were made to compute accelerator wake fields on the fly (in contrast to post processing for wake fields). These modifications made it possible to derive the wake fields for model e-beam kickers in a extensive single run rather than requiring two runs, storing massive quantities of data, and post processing. While the product of our test was just a serial DSI time domain code, the software had new physics not even in the original code. We able to build and test real accelerator problems not possible prior to this project.

We successfully modeled several AHF Kicker structures and achieved over a two order of magnitude reduction in the number of required unknowns compared to earlier TSAR models. We had turn around times ~~that required~~ requiring hours rather than weeks of CPU time. From many perspectives we were very pleased with the results. Very early in the project we had already achieved many of our objectives with regard to AHF accelerator components. However, this was just a prototype version; we next turned our attention to a hybrid mesh, massively parallel version.

~~We learned a lot from our first version.~~ Most of the time the abstractions worked well and allowed us to ~~the~~ build a code using on the order of one third the effort. (Here "one third the effort" is just an educated guess based on our experiences.) next page

The biggest limitation of the TIGER-I scheme was that there were too many layers of software, each of which occasionally had to be punctured to allow access to lower levels. The syntax that developed over this evolution was awkward and unwieldy. Decisions made early in the development of TIGER-I made it difficult to incorporate hybrid meshes (structured or unstructured). In addition there were unacceptable memory overheads and the software ran slow. The decision was made to start over with a very different model of Mesh management system in our next version.

From a flexibility and extensibility point of view, we found the TIGER-I scheme limited. TIGER-I consisted of containers that managed large collections of Mesh components. There was and Array_of_sets container where nodes, edges, faces, cells where placed. From a finite difference or finite volume approach we found the rigid nature of our data layout too restrictive. We knew that some of these concerns would go away as we added material, interface, field, algorithm, etc... classes, but a many of the limitations where solely based on the lack of quick, transparent access to data. For example, suppose we wanted to apply a particular algorithm to all the edges of a problem space that had one node on the surface of two materials and the other node within a material. The only solution was to loop over all the edges and check each edge's data. In general, we had no way of iterating over data, (i.e., all cells containing the material copper), without iterating over all the entire mesh. What was most disconcerting was that this general way of accessing the data was by far the most common way we wanted to access the data. In a finite difference scheme there often is no real mesh. The mesh is implied by position within an array or by formula. A very common method is to tag various mesh components with data and iterate over the data independently from the mesh. Much of the complexity of the existing software was created to specifically address this need and our Mesh management system did not handle this well. The approach we ended up taking was to iterate over all the edges or faces and apply~~ing~~ the appropriate algorithm based on looking at the data. This was limiting in that it created a host of special cases especially when the questions determining the algorithm were of compound nature and/or involved indirect nearest-neighbor topology considerations. This approach did allow us to build a code but we wanted to address this limitation in our next version.

Organizing the data based on data and data associations could alleviate our general data access and iteration needs. Also, moving the data abstractions down to very low levels (i.e. nodes, edges, faces, cells) would reduce the required complexity of management systems and provide the object-oriented freedom we were striving for. The question was how to accomplish this. It is very well known in object-oriented environments that if performance is an issue ~~that~~ then abstractions should not be down at the lowest levels. We were already running into performance concerns in TIGER-I due to the very light weight nature of finite difference and finite volume techniques. Typically, we have say two multiplies and two additions per component of work to perform for every memory fetch that is required. This small CPU effort per data item is the primary reason why the execution of finite difference and finite volume schemes are typically always memory starved. So the notion of having hundreds of millions of objects being constructed, each with its own data and methods, rather than accessing a few tens of large containers sounded appealing but appeared to be impractical. Both the memory footprint on a per object basis and the CPU required to accomplish this seamed daunting.

We learned a lot from out first version. An overview of our experiences to date were that while that object-oriented programming using C++ was very powerful it had some large limitations that we had to overcome on the project. The C++ compilers were deficient but were improving. Object-oriented programming was no silver bullet, it was going to be a lot of design and hard work. Memory and CPU efficiency concerns influenced our design and implementation far more than we would have liked. Our experience to date was that object-oriented software development was no faster than FORTRAN software development when the core class abstractions needed to be modified. Perhaps it was even slower due to the tremendous extra effort required to design the classes. However, the end product was much more flexible and C++ was inherently better suited to adding behavior extensions than FORTRAN.

Design Overview of TIGER-II

In TIGER-II the basic philosophy of how the mesh was to be stored was changed. Instead of storing indices into tables as the ups and downs of an Entity, actual pointers to the entities are stored. This removes a step from the process of moving from one Entity to another, and it was hoped this would bring an increase in speed to the code. More fundamentally it facilitated the use of pointers to real objects rather than having an index into a container where raw data lived. Another philosophical difference is that instead of having a single integer tag to describe everything there is to know about an Entity, we wanted to be able to tag entities with arbitrary Attributes. We wanted to add the capability to efficiently access the data in user defined ways. If possible we wanted to have more polymorphism at node, edge, face, cell level and not up at the container level. In addition, we wanted to have hybrid meshing and parallel processing built more explicitly into the code.

Another difference between TIGER-I and TIGER-II architecture is that in TIGER-I entities were divided into cells, faces, edges, and nodes, while in TIGER-II there are only entities. Entities have Attributes, some of which are local (such as position, or material), while some are shared (such as whether an Entity is a node, edge, face, or cell). In

TIGER-II entities were stored in two parts. Every Entity could have local data which was stored in an Entity specific piece of memory, but every Entity also had shared data (Attributes and the number of ups and downs). The Attributes of a group of entities were collected in one place (called a Domain), which was pointed to by every Entity that shares those Attributes. The local data (including the pointers to the ups and downs) were stored in Noncontiguous arrays. For unstructured meshes there was one Noncontiguous array for each Domain, while for structured meshes it was possible for many types of entities without local data to live in a single Noncontiguous array and point to different Domains.

As stated above, we wanted to have a more general Attribute tagging ability in TIGER-II. We wanted to be able to tag any Entity with any Attribute. To this end we built a number of Attribute data classes, all of which were specializations of a root Attribute class. Any Entity could have one Attribute from any of these classes (although some were contradictory: it doesn't make sense to have an Entity that has both Node and Cell Attributes). The Entity Attributes were pointed to by the Entity's Domain. Attributes ~~can~~ could have three types of data in them: local, shared, and computed. Local data are specific to an Entity, and there must be room allocated to store the local data when the Entity is created. Shared data resides in the Attribute and has no memory overhead. Computed data is generated every time the appropriate Attribute method is called and has no memory overhead (especially useful when trying to determine the ups, downs, and positions of entities in a structured mesh). Adding an Attribute to an Entity causes it to change Domains to one that has that Attribute in addition to all of the current ones. If a Domain with the Entity's new Attribute set doesn't exist, a new Domain is created, and a new Noncontiguous array (with spaces sized for the new Entity) is generated to hold the Entity. The Entity is moved and the local data for its Attributes are put in the appropriate places in its new place.

See Figure 1 for a simplified picture of the memory layout.

A great deal of effort went into making sure that TIGER-II would be able to use hybrid meshes. Hybrid meshes are meshes that are made of a number of sub-meshes, any of which may be structured or unstructured. Structured meshes have the potential to greatly reduce the memory required to store them because all of the mesh parameters (positions, lengths, areas, directions, and volumes) can be computed from a very few values. Multiple unstructured meshes allow different mesh generators to be used in different parts of the problem (an advantage in some circumstances). To simplify the coding we decided to only hybridize unstructured meshes. In order to hybridize structured meshes (either to other structured meshes or to unstructured meshes), they had to be wrapped in an unstructured skin. Reconciling meshes to one another also turned out to be a difficult problem on single processors, as the individual meshes each had their own node numbers which made it difficult to find corresponding nodes, edges, and faces.

The overall structure of TIGER-II looks like this:

```
Supermesh
  Mesh
```

```
              Factory
    Domain    Entity    Attribute
               Array
```

At the bottom once again is the Array class. The STL libraries still did not perform as we needed them to in order to make best use of memory and CPU. The Array classes were very similar to those developed in TIGER-I and little effort went into changing them.

The Entity, Domain, and Attribute classes were all interdependent upon one another. An Entity contains a pointer to a Domain, pointers to the ups and downs, and whatever local data needs to be held by its Attributes (pointers to which are held in its Domain). The Attribute classes tell the Entity what it is. There are broad groups of Attributes (such as Topology, which tells an Entity if it is a Node, Edge, Face, or Cell, and Material, which tells an Entity if it is a Maxwell_Material (and if so if it is copper, aluminum, etc.)). Any Attribute pointed to by an Entity's Domain points back to the Domain. The Domain holds all of the shared information pertinent to a set of Entities (all of the Attributes they contain, how many ups and downs they have, how many there are, and where they start in memory). If any of the Entities held in a Domain change in any way (adding an up or a down, adding or cutting an Attribute, etc.) then that Entity is removed from the Domain's care, and handed off to another Domain. If there is no Domain currently existing that has the Entity's new Attribute set or up or down count, the Factory creates a new Domain.

The Factory creates Domains and Entities. A request to create an Entity arrives at the Factory with a set of Attributes that describe it and the desired number of ups and downs. The Factory looks through the records of previously created Domains to find one that matches the Attribute set and up and down counts. If it is unable to find a Domain to match, it creates one, as well as allocating a Noncontiguous_array in which to hold the Domain's Entities. After finding (or creating) the relevant Domain, the Factory creates an Entity in the Domain and returns a pointer to it.

The Mesh is the object that asks the Factory to make Entities. The Mesh has two jobs: load a mesh from a file, and provide a set of Entities based on a user supplied filter. In loading, for example, an unstructured mesh from a file, the Mesh class uses the Factory to first create the Entities that represent the nodes. These Entities have a Node Attribute (which contains the node position information), and, initially, no ups or downs. As they are loaded, their global node numbers and a pointer to their corresponding Entity are recorded. After the nodes are loaded, the cell records are read. These records contain a material number and a list of global node number from which the cell is constructed. The order of the cell's global node numbers gives the connectivity of the mesh and specifies the edges and faces that need to exist before the cell can be created. Some of the edges and faces may already have been created, and we have to use the global node numbers to access the cell's Node Entities, and check to see if they have the appropriate edges shared between them. If they don't, the edges have to be created (by calling the Factory to create an Entity with an Edge Attribute, two downs, and no ups), and connected to the appropriate Node Entities (which have to have an up added to them, so they move in memory). Once the Edges are in place, the Faces must be either found or created (and connected the Edges (which move)). Finally, the Cell can be created (and connected to the Faces (which move)). Loading structured meshes is less straightforward.

The Mesh's second job is to give the user a set of Entities based on a user-supplied filter. By specifying a filter, the user asks for the Entities that have particular Attributes and up and down counts. So if, for instance, the user wanted to see the exterior of the mesh, he would build a filter that asked for all Face Entities with one up (one cell). When coupled to a visualization tool, this would allow inspection of the mesh to determine if there were loose faces flapping around on the interior. Filtering and visualization where of great utility in debugging.

The Supermesh is the object that orchestrates the loading of the multiple meshes and reconciles them to existing together. The Supermesh reads a supermesh file which tells it what kind of mesh files to load and the associated mesh file names. Once each of these meshes is loaded independently, the Supermesh reconciles them. Reconciliation works as follows: for every Mesh, every exterior Face is examined to see if all of the Nodes of the Face are duplicated in any other Mesh. If they are, and they make up a face, then the Edges of the Nodes are pointed to the duplicate Nodes, and the Nodes are deleted. The Face is pointed to the duplicate Edges, and the Edges are deleted. The Face's Cell is pointed to the duplicate Face, and the Face is deleted. There are many special cases and the process is quite involved.

TIGER-II removed many of the limitations we found in TIGER-I. The user could ~~can now~~ define, in a limited sense, the data organization making it possible to say something like "Iterate over all edges having Attribute "XYZ" connected to faces containing 3 edges". This was a runtime question requiring no new code to be written. This freedom allows one to trivially attach data, Attributes, algorithms, etc… over arbitrary collections of Entities. The abstractions have been moved down to the Entity level so it ~~is~~ was possible to ask an Entity ~~a such as~~ questions about length, position, material independently of what container the Entity lives in. TIGER-II is capable of loading and ~~automatically~~ stitching together an arbitrary number of structured, warped, extruded, revolved, and/or unstructured ~~mixed~~ element meshes automatically. Via a simple Graphical User Interface (GUI) the user can apply a filter string to query the resultant mesh. TIGER-II supports hybrid meshes in that a single unified interface exists ~~to iterate~~ and query the mesh independent of the underlying mesh type.

TIGER-II provided a huge jump in generality and had ~~a much~~ a markedly improved interface over TIGER-I. TIGER-II also allowed limited massively parallel unstructured meshes. (The massively parallel hybrid meshes implementation was not completed.) ~~The user was~~ presented with a nice clean interface but the bookkeeping ~~we~~ abstracted behind the interface ~~was very~~ became very complex. In fact, the processes of structured mesh loading and of mesh reconciliation were so involved that they made further progress on the code extremely cumbersome. Structured meshes had to live at known memory locations and needed to be wrapped in an unstructured layer so different mesh types could be stitched together. When multiple structured meshes came together the same mesh Entity had to live at multiple known locations. Extruded and revolved meshes were truly hybrid in that of there topology

some of their topology was computed and some was pointer references. So when a user invoked some action that caused an Entity to relocate the bookkeeping involved with abstracting the bookkeeping ~~was~~ tremendous.

that

Also, we expended a great deal of our effort trying to keep our design and implementation efficient along the way. However, even though we were able to design a class structure that had very few virtual functions, performance was starting to become a problem. Actual performance is problem dependent but roughly speaking our final TIGER-II implementation is on the order of 10 times slower than its tuned FORTRAN. To be fair TIGER-II is dealing with a complexity far surpassing the ~~required~~ bookkeeping for DSI or TSAR and most of the performance difference ~~is~~ associated with this complexity. Note these times are only approximate ~~and~~ only include the Mesh construction times. The overall CPU time differences much less, perhaps (2x-3x). The required memory was similar but slightly larger ~1.4X (TIGER-II versus FORTRAN DSI EMCC research version. Most of the versions of DSI actually require more memory than TIGER-II but this is due to making conservative array size estimates. The 1.4X was calculated some time ago using theoretical DSI minimum required memory. The details have been lost and are only included to give some simple sense of memory overhead.) 15

On one hand, future physics packages would be spared much of this complexity, and after all, the insulation of the physics from the bookkeeping was what we were trying to achieve. There would hopefully be many physics packages but we only needed one bookkeeping package and this was essentially written. On the other hand, through our research we discovered ways to significantly reduce the complexity of our implementation and ways to further improve the memory overhead and flexibility of our interface.

A major redesign and implementation of our abstractions would be a huge undertaking so late in the project even if we used the latest in modern software techniques. However, the expected increase in capability, flexibility, maintainability, and performance, as well as the expected decrease in complexity would dramatically improve this research effort. It was a very tough decision to make.

For above mentioned reasons (as well as the Attribute tagging not being as general as was deemed necessary, difficulties doing mesh reconciliation over multiple processors, and lack of thread safety), it was decided that TIGER-II would be discontinued, and TIGER-III was brought into being. Having an unplanned third version start so late in the project meant that we would not have the time to put new physics into the last version to demonstrate its capabilities but the core of our research thrust was to investigate flexible abstractions to insulate the physics from the bookkeeping.

Design Overview of TIGER-III

TIGER-III was designed to alleviate the deficiencies of TIGER-II. One of the implementation complex and most time consuming (and never actually completely debugged) parts of the coding for TIGER-II was the reconciliation routine. TIGER-III did away with the necessity of after the fact reconciliation by examining each incoming node and determining if it had already been loaded from a previous mesh file. TIGER-III tries to address bookkeeping issues in a more fundamental way than did TIGER-I or TIGER-II. The following sections describe an overview of three API libraries developed in the project. The three appendices are large machine generated text documenting the current state of TIGER-III's API. The automated appendices are far from polished and have many grammatical, spelling, syntactical, or outdated comments. The appendices are included for completeness and to help supplement many details missing in the following sections.

The computer science aspects of this project are not called out specifically in this document. It is the authors' intent to describe the design patterns used in TIGER-III in a computer science publication after the implementation of three libraries is finished. Here we will just give an overview of the large packages that constitute TIGER-III's API.

Container Classes

Large arrays of data are ubiquitous in scientific computing. Older programming languages such as C and FORTRAN had built in support for fixed-length arrays. However these arrays were not allowed to grow or shrink in size, and they were not "safe" in the sense that it was possible to accidentally over or index these arrays. The C++ language allows the user to create user-defined array classes than can grow or shrink in size and can be made safe. In addition, other general-purpose container classes such as trees, lists, maps, etc. can be easily constructed in C++. Through the C++ template mechanism these containers can hold arbitrary objects. This facilitates software re-use and promotes more robust code.

The Standard Template Library (STL) is a C++ library of general-purpose containers such as arrays, lists, maps, etc, and associated algorithms that operate on the containers. This library is now standard with most commercial compilers, and free versions are available on the Internet. Unfortunately in the early phases of the TIGER project the STL was still not 100% standard and it was deemed too inefficient for scientific applications. Hence, we decided to develop our own hierarchy of general-purpose container classes and algorithms. In addition we developed some extremely useful general-purpose container classes that are not found in the STL. Also, much effort was made to make all of the TIGER-III container classes thread safe.

The TIGER-III Array classes are arranged into a hierarchy that uses both inheritance and composition. The lowest level classes are the Heap array and Stack Array classes. These classes are simple fixed-length arrays that live on the heap and stack respectively. These arrays use the [] operator for access and thus mimic the built-in array syntax. These arrays do check for over and under indexing. It is not intended that these classes are used directly, rather they are used to construct other more sophisticated arrays.

The Adjustable array is an array that can grow or shrink in size automatically. Adjustable arrays are derived from Stack array classes. These arrays also use the [] operator, but they do not check for overflow since they are allowed to grow in size. The Adjustable array works by keeping track of the number of items in the array. When the Adjustable array is required to grow (either automatically or by user demand) new memory is allocated, the data is copied from the old memory to the new memory, and the old memory is freed. This operation of allocating and de-allocating memory has some disadvantages. First, memory can become fragmented, which will result in the program running out of large blocks of memory. Secondly, if data pointers outside of the array were pointing to data inside the array, after the memory reallocation the pointers are invalid (the well-documented dangling pointer problem). A memory pool that obtains large system blocks of memory is used to help circumvent memory fragmentation. Adjustable arrays are good for small (few) to medium (few hundred thousand) elements. Huge arrays should be built using Noncontiguous arrays.

The Noncontiguous array is adjustable but items in the array never move, hence the Noncontiguous array does not suffer from memory fragmentation or dangling pointer syndrome. To the user, a Noncontiguous array behaves exactly like an adjustable array. The Noncontiguous array contains within it an Adjustable array of blocks. The Noncontiguous array works by allocating small non-contiguous blocks of memory as needed. Additional bookkeeping is performed internally to keep track of the blocks. Due to this bookkeeping, a Noncontiguous array is slightly less efficient than a standard Adjustable array. The Noncontiguous array can not be found in the STL, and the Noncontiguous array is an essential element of the TIGER-III Entity-Attribute data structure.

A Reference Noncontiguous array is a specialization that hides data common to all items in a block in the block header. This is to minimize memory usage by eliminating redundant storage of like data. In conjunction with the Memory Pool, large blocks of data are stored at specially aligned memory locations. This allows the fixed memory of all our Entities to be 2 pointers instead of 3, resulting in 50% savings of memory overhead. Also, by storing extra information in the header of each block which is placed at specially aligned memory locations, running indices can be stored. This hidden data allows us to efficiently compute global contiguous numbers for every Mesh Entity and Attribute across all processors storing no local data within an Entity. The use of the Reference Noncontiguous array is an essential key to TIGER-III's overall memory and CPU efficiency.

A Sorted array is an array that is always kept internally sorted. The sorted array is derived from Adjustable array. The array provides random read access but not random write access. The only write access is an insert() method that inserts the data item in the appropriate place using a binary search algorithm.

The Container class is a decorator class[1] used to add functionality to all of the above array classes. Specifically, the Container adds relational operations to the arrays and it facilitates the copying of data from one type of array to another type. In addition, the Container decorator adds pack() and unpack() methods required by the communication class.

A Registry is a Sorted array of Pairs, where a Pair consists of a Key and of a Data item. The Pairs are sorted according to the value of the Key. Common use of a Registry is to associate character strings or integers with pointers to functions or other objects.

A Balanced Binary Red-Black Tree (BBRBT) is a data structure where the data items live on the "leaves" of the tree, with the leaves organized by a series of branches that split into two and all originate from a single root. The BBRBT behaves like a Sorted array, it is the internal implementation that is different. The Sorted array suffers from the same memory movement problems as the Adjustable array. The BBRBT minimizes data movement, which can be important for maintaining very large sorted containers. Insertion and deletion of items approaches a constant time for large data sets.

An Oct Tree is a three dimensional variation of a binary tree. An Oct Tree is an ideal data structure for storing (x,y,z) triplets. In simple terms, space is divided up into cubes and triplets are put into the appropriate cube. Determining if a given (x,y,z) triplet exists in an Oct Tree scales as logorithmeticaly, hence it is fairly efficient. An Oct Tree can be memory intensive, and special care was used to implement the code in a memory conservative fashion. The Oct Tree is an essential part of the Mesh class.

To summarize, the TIGER-III framework has several standard container classes that are used thought the framework. These classes all use the C++ template mechanism. Much effort went into to making these classes efficient both in terms of memory usage and in CPU time. These container classes all use the Memory Management class and the Monitor/Mutex class that are described below.

Miscellaneous Utility Classes

The TIGER-III framework provides some low-level utility classes. In simple terms, a utility class is a class may be used by all the other classes and is treated as if it were an intrinsic part of the development environment.

The String class is used to represent character strings such as "Hello, World". Most modern C++ compilers come with a built in String class, but the built in String class is not particularly efficient. The TIGER-III string class is basically an array of characters, with some additional member functions for relational operators and type conversion

The Memory Management classes (Memory Pool, Memory Stamp, and Memory Manager) are used to manage the allocation and de-allocation of memory and to gather statistics on memory usage. These are low level classes are not intended to be used by the user, instead these classes are used inside of the other TIGER-III classes such as the array classes. The memory is allocated in huge blocks of data and is split into smaller data chunks that are specially aligned. A user can mask the least significant bits off of the memory and jump to that memory location that contains extra data that all items in the Memory block share. Common uses are to store pointer addresses that every item in the block has in common or to store running block offsets that can be used in conjunction with an item's raw address to obtain unique identifiers. These methods are used very frequently throughout TIGER-III and often facilitate huge memory savings.

The Monitor, Mutex and Thread classes are for use in a multi-threaded environment. These classes are used to create threads and lock down objects so that only one thread can access them at a time. In simple terms, a thread is a subroutine that is running independently and in parallel with the main program. Most modern non-scientific applications make heavy use of multiple threads, for example in an e-commerce system one thread may me running the graphical user interface, another thread may be writing data to disk, while another thread may be performing data base queries. This type of parallelism is often called task-based parallelism because each thread has a well defined task is distinct from the tasks being performed by other threads. This is in contrast to SPMD parallelism were each processor is performing the same task on different data. While a small portion of the TIGER-III framework currently uses multiple threads, the Monitor and Mutex classes were developed mostly to help enable research on multi-threaded algorithms for use on clusters of SMP computers.

Communication Classes

The TIGER-III framework was designed for parallel execution on both shared memory and distributed memory computers using the single-program multiple-data (SPMD) paradigm. In this paradigm a single program is executed simultaneously on multiple processors and each program has direct access only to a portion of the total amount of memory. In other words, data such as the mesh, the fields, etc. is distributed across all of the processors. This approach is also sometimes referred to as domain decomposition. Note that some parts of the TIGER-III framework use multiple threads of execution, this is a different type parallelism and is discussed in another section.

In the SPMD paradigm the processors often need to exchange data. This communication is referred to as message passing. The TIGER-III framework uses the Message Passing Interface (MPI) library for all message passing. The MPI provides functions for sending and receiving messages, for synchronization of processors, and for collective operations (summation, maximum, etc.). The purpose of the TIGER-III Comm classes is twofold: 1) to collect all MPI function calls into a single file for ease of maintainability, and 2) to facilitate the communication of arbitrary C++ objects. The former objective is both standard and trivial; the latter objective was non-trivial due to the fact that MPI has a C (or FORTRAN) interface and is unaware of C++ objects. The starting of a MPI process, synchronization, and collective operations are supported by the TIGER-III Comm class, but since these are fairly standard operations, they will not be discussed further. The communication of arbitrary C++ objects is discussed in more detail below.

It is important to review some basic aspects of C++ classes and objects. The basic C++ data types such as int, float, double, char, etc. are considered to be intrinsic classes. An instance of a class is an object. The MPI library is capable of communicating these intrinsic objects, or arrays of these objects. A user-defined class is composed of intrinsic classes and/or other user-defined classes. The TIGER-III classes such as Entity, Attribute, Species, etc. mentioned in other sections of this report are user-defined classes. It is these user-defined classes that MPI does not know anything about, hence the main purpose of the TIGER-III Comm class is to facilitate the communication of other TIGER-III classes.

It is undesirable for the TIGER-III Comm class to have detailed knowledge of the other TIGER-III classes; data hiding is a basic tenant of object-oriented design. If the TIGER-III Comm classes did have detailed knowledge of the other TIGER-III classes the software would not be extensible or scalable. But how can the TIGER-III Comm class be "in charge" of communicating all the various TIGER-III objects if it has no knowledge of what is inside of these objects? The following is an example:

Consider class A which contains an int and a float, and class B which contains a string and a pointer to an object of type A: (the following is pseudo code)

```
class A {                   class B {
      int x;                      char name[6] = "Foo";
      float a;                    A *ptr_to_an_A;
};                          }:
```

Let's assume that on processor 1 we have a properly initialized object of type B named Foo, and we want to send Foo from processor 1 to processor 4. We have an instance of the Comm class that is responsible for communicating Foo. How does the Comm object know what parts of Foo to communicate? Should a shallow-copy (send the pointer) be performed or a deep copy (send what the pointer points to) be performed? An additional complication is that the MPI library cannot directly communicate objects of class A or B even if the Comm class knew what parts to communicate. The solution to this dilemma was to require every TIGER-III object that may be communicated to have both a pack() and an unpack() method. These methods pack/unpack the object into/out-of a buffer. The Comm class then sends and/or receives the buffers. This process is recursive; the recursion terminates when the Comm object is given an intrinsic object that it knows how to deal with. The Comm class performs the pack/unpack of intrinsic objects, since it is not possible to endow an intrinsic class with a method.

To complete the example, the following is the sequence of events that enable the communication of Foo from processor 1 to processor 4:

The Comm object attempts to communicate Foo, but since Foo is not an intrinsic object the Comm class tells Foo to pack himself into a buffer; the Foo object packs the character string name into the buffer (a character string is an intrinsic class). Foo knows that he needs to perform a deep copy of ptr_to_an_A, so he de-references the pointer and tells the object of type A to pack itself into the buffer; the object of type A packs the int and the float into the buffer, as int and float are intrinsic classes; the Comm object sends the buffer to processor 4; 5) on the receiving processor the above process is repeated in reverse order.

To summarize, TIGER-III objects that are to be communicated must have pack() and unpack() methods. The Comm class uses these methods to pack and unpack the object into/out-of a communication buffer. All communication is done using MPI_PACKED data type. The C++ feature called template specialization is used to implement the Comm class. The Comm class is an independent utility that can be used by other parallel programs; it does not depend upon the rest of the TIGER-III code. The Comm class is thread safe, which is essential if two or more threads are performing MPI communication.

Compared to other approaches of MPI communication that use the built-in ability to create and communicate static structured data types, the pack/unpack approach is significantly more general at the expense of performance. The performance penalty is due to the packing and unpacking of data into and out-of buffers. Since few data structures in TIGER-III are static, efficient communication of static data structures was not given a high priority. In other words the majority of TIGER-III objects may grow and shrink during execution of the program and the above approach remains a valid communication scheme for these objects.

Entity-Attribute Classes

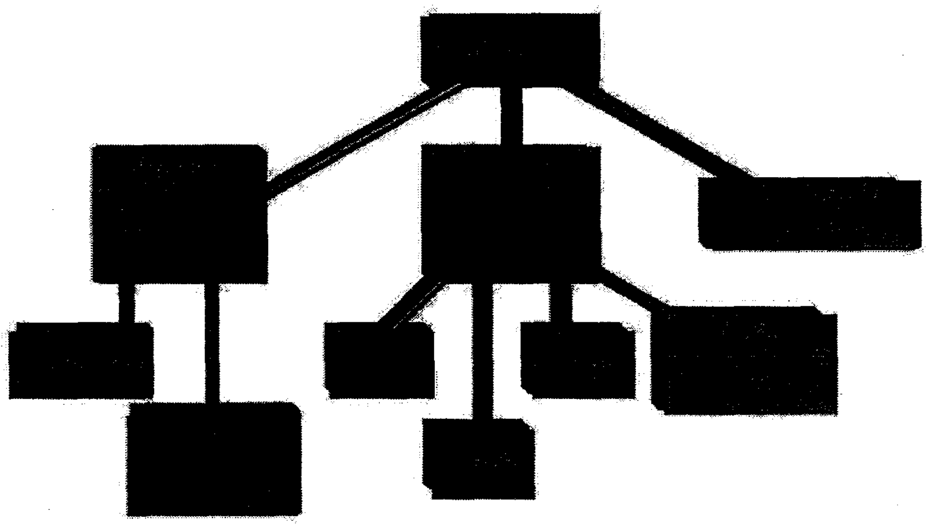
The key development in TIGER-III is the creation of the Entity-Attribute library. It provides an abstraction between objects and their type. As such, a large number of these lightweight objects(dubbed Entities) can be created and change their type at run time. The overhead per Entity is very low (specifically, two pointers), which allows us to model each logical object in a system, such as a node, edge, face, cell or particle, as an Entity. While Entities were designed to model the topological features of a mesh, there's nothing limiting them to that role. A user could use Entities to model neurons in a neural network for instance.

An Entity can contain one or more Attributes. As Attributes are added or subtracted the Entity changes behavior. For example, consider an Entity that has only one Attribute, say Material. A user can't ask this Entity for its length because it has none. But if the user adds an Edge Attribute at run-time and then asks for the Entity's length he or she will obtain the correct solution. The fixed memory overhead for an Entity is exactly two pointers plus any local data. So a Node Entity that had a position and no other data would have 2 pointers and 3 floats (or doubles). An Entity does not, in general, have to change memory location even when new data is added or subtracted which changes the Entity's size. Of special note is that every Attribute attached to every Entity has a unique Global IDentifier (GID) which is contiguous across all processors but costs no local memory overhead. (There is one exception to this, all Entities owned by one processor but referenced by other processors must store a local Global Entity Identifier GEID number when residing on the non-owning processor). This means that if a user tags a random set of Entities with say copper. The user can iterate over all copper Entities and ask a given Entity what is your copper GID and get back a number that costs no local memory. The actual overhead is one word per large group of Entities (per processor) although each Entity has a local copper GID unique to it. How this works complicated and will be explained later on in the discussion.

An Attribute is a class the user builds in software. Typical examples of Attribute classes are Nodes, Materials, Boundary Conditions, Outer Surface, Topology, Fields, etc... The user defines the data and the bookkeeping is abstracted from the user. Attributes replicate themselves across processors without user intervention. Attributes are discussed later.

A collection of Attributes makes up an Entity. All Entities with an identical set of Attributes make up a Species. The user can iterate over Entities, Attributes or Species.

The data and behavior of each Entity depends on what Attributes it is tagged with. In essence, Attributes provide type for Entities. A user can change a given Entity's data and behavior at run time by modifying what Attributes that Entity is tagged with. Attributes are defined by a type hierarchy, which starts with a root Attribute. Various Attributes inherit from this root Attribute, and more Attributes can inherit from those. This is all done at compile time though basic C++ inheritance. These Attributes (called the Type Attributes) define the data and behavior for Entities tagged with an Attribute of that type. Attributes that have the same class name (Type) but different data are also distinguishable by their Kind. So an Entity can change its behavior just by changing the data associated with one or more of its Attributes. This happens at run-time. If an Entity changes its local data only the behavior of the Entity is changed. If an Entity changes data that is common to other Entities then all Entities sharing that common data will immediately change behavior. The behavior is similar to the State design pattern [Gamma, et. al., p. 305].



There is one instantiation of every Type Attribute that is created in the static initialization sequence before main(). This object gives the user access to all information regarding that

Figure 1: Example Type Attribute Hierarchy

Type Attribute and the ability to create specific instances of this Type Attribute at run time. These specific instances of Attributes called the Clone Attributes. Clone Attributes define the value of data common to all the Entities that the Clones are tagged to. The clones are created by a Factory [Gamma, et. al., p.107] to insure that no two clones have the same Common Data values. Doing so keeps the number of Attributes to keep track of to a minimum and reduces the memory overhead of the system.

The Clone Attributes can be created either though the use of an Attribute_db file or user at run time. The Attribute_db file is just an ASCII flat text file that defines Clone Attributes that will be used every time the user's code is run. This frees the user from having to make numerous calls to create these common Attributes in their code. The entries in the Attribute_db file consist of a Clone Attribute name, the name of the Type Attribute that it is a clone of and a list of data definitions that specify the name of the data item (which must match the name defined for that data item in the declaration of the Type Attribute), the data type (Common, Local, Computed, etc.), and the value or function name for that data item. The Attribute_db file should be static for a given code, as the code will depend on these Clone Attributes having being defined before the code begins execution. Clone Attributes can be added at run time but for convenience we allow the rest of the software to depend on the Clones in the Attribute_db file being setup.

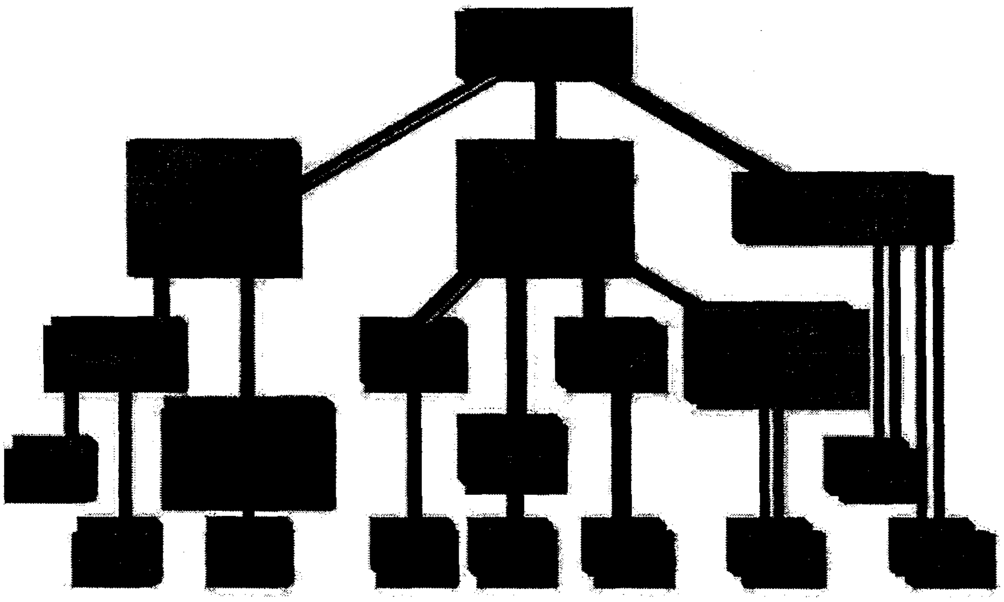


Figure 2: Example of Clone Attributes created from the Type Attributes.

Entities can be created or modified to have any combination of Attributes, including two Clone Attributes of the same Type, if that Type allows it. This is useful for cases such as the interface between two materials, where the Entity should be tagged with both materials. This requires some special handling on the part of the programmer implementing the physics, as the programmer can no longer just ask what Clone Attribute of that Type the Entity has. Instead the programmer must ask for a list of Clones (which may just be a list of one), and iterate over it. An Entity can not have more than one copy of a Clone Attribute, as this is not necessary. If an Attribute can have multiple data values, then that data should be stored in an array, not though multiple clones.

Attributes can have three basic types of Data: Common, Local and Computed. Common Data is data that is defined as being the same for all Entities that have that Clone Attribute. Local Data is data that is local to an Entity. Computed Data is, as the name implies, computed on the fly for that Entity by a user defined function. There is a forth data type called Local Array, which is really just a specialization of Local Data allowing an Array of data values to be stored in each Entity with that Attribute in standard C style (that is, accessed through the C [] operator given a pointer, and without any bounds checking). All of these data types are specializations of At_value, which can be used by a class as the data type when it is possible that different Clones of a given Type Attribute may store their data differently.

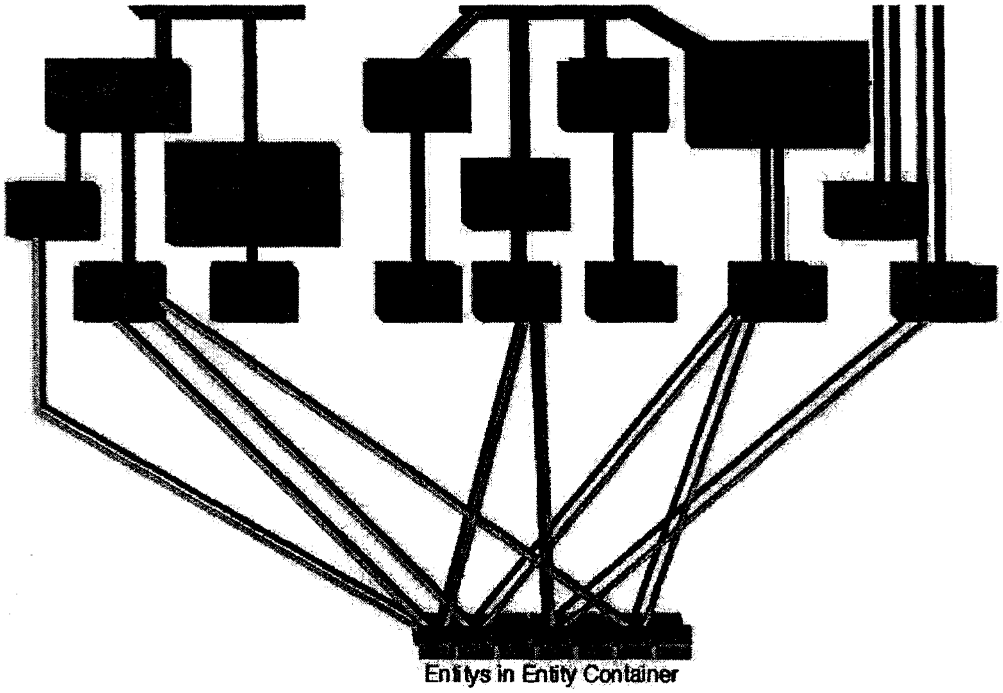
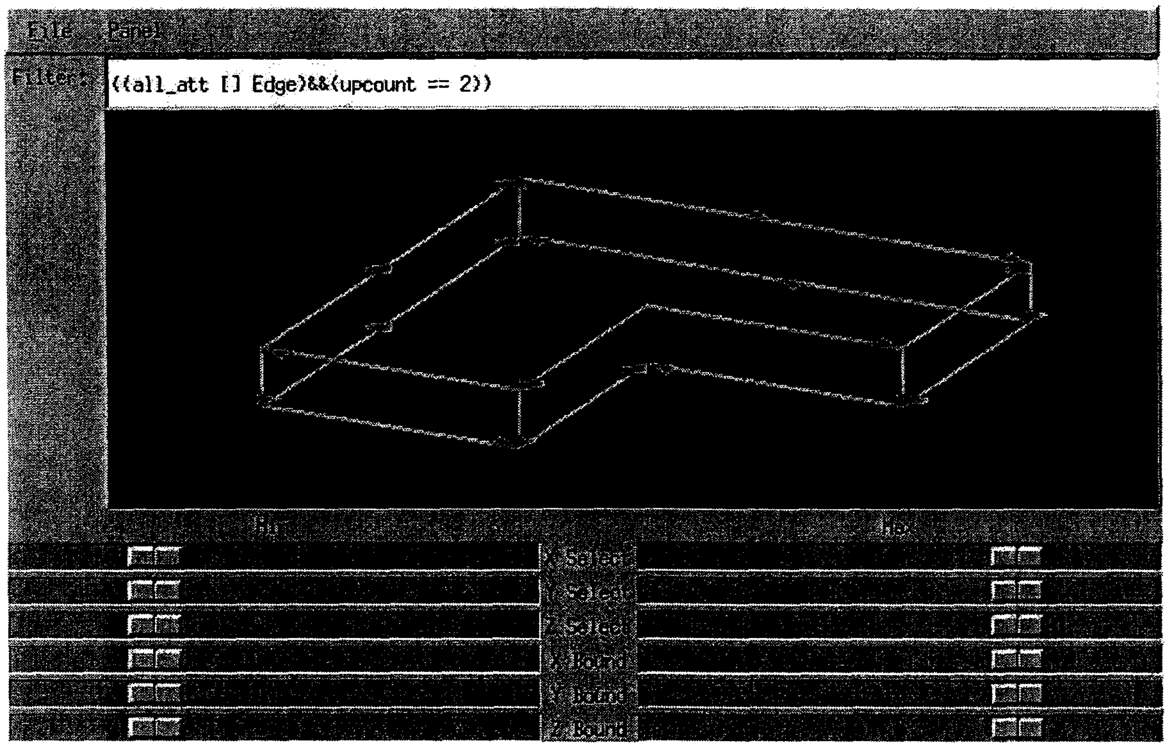


Figure 3: Entities with Attributes. Note that multiple Entities may point to the same set of Attributes and that an Entity may point to multiple Clone Attributes of the same Type.

Given a large collection of Entities and their Attributes, the most common operation that a user will want to perform is iterating over a group of Entities with common Attributes. For example, looping over all the Edges in a problem or looping over all the Entities tagged with both the Type Attribute Boundary Condition which have only one up in their Topology Attribute This is accomplished with a Filter. The user can create a Filter with a string at run time. This allows the user to either create or hard-code a string in the code for processing, or enter the string though a user interface at run time. That allows the user to pick the data they want to view or manipulate it by hand without having to change a line of code. The authors have found this feature to be one of the most useful ways to debug the Mesh class during its creation. However, one can use the same procedure to apply specific algorithms to specific portions of the mesh.



Filter: ((all_att [] Edge)&&(upcount == 2))

Figure 4: TIGER-III GUI with Filter line. This particular filter command finds all edges with 2 topological ups.

The hierarchy of all Type and Clone Attributes is the same in all processes in an MPI job. As the Type Attributes are created at compile time and their static instantiations are created before main() is called, these are automatically the same in all processes before Communication is even initialized. Once the program has started, a specialization of Factory called CommFactory is used to create the clones. The CommFactory will create a unique instance of every Clone Attribute in each process.

The CommFactory works asynchronously through the use of threads running in every process. When a Type Attribute is initialized in a multiprocessor environment, it starts a thread to handle the creation of its clones in that process. Then, when the user requests a Clone of that Type Attribute, the CommFactory first checks if it already exists in that process. If it doesn't exist, the CommFactory requests the given Clone Attribute from a designated process (usually process 0). This serves as a deliberate serialization so that if two processes request the same Clone Attribute at the same time, when the designated process is done cloning the first request, it knows there's nothing to do for the second request and only one instance of that Clone Attribute will be created (as it should be). If the designated process actually needs to clone the given Attribute, then a message is sent to all other processes with the argument passed to CommFactory and the new Clone Attribute is simultaneously created in all processes. An acknowledgement is then sent back to the process that originated the request so that the new Clone Attribute can be found and returned to the user.

The method described above may not work well depending on the underlying thread implementation. The problem manifests itself as the number of Type Attributes increases. There is one thread per Type Attribute, hence as the number of Type Attributes increases, the number of threads to assist the cloning operation increase. These threads spend most of their time idle, waiting for a message telling them to create a clone for the Type Attribute they support. Some Machines, such as the SGI (IRIX 6.5.5) show no degradation as the number of threads waiting for messages increases. Other machines though, such as the Compaq Alpha (Digital UNIX 4.0), decrease in performance as the number of latent threads increases. Both machines are four processor machines running the MPICH shared memory message passing library. Hence, the threads aren't truly latent; they're polling the MPI library to see if they have a message waiting which requires that they lock down the MPI library as MPICH isn't thread-safe. The problem may be solved though more careful attention to thread scheduling (right now all threads are running with the same priority with the default scheduler), or by using a thread safe MPI library so that the threads could perform blocking receives and hence be truly latent. Alternatively (what may be the best fix), the architecture could be changed such that one thread handles the cloning for all Type Attributes.

All the Entities in a program (or at least in one process) live in an Entity Container. A given Entity is uniquely identified by its position within this container. Within a single process, the address of that position is sufficient and is the most direct, efficient and hence frequently used way to access an Entity. Across all processes, Entities are assigned GEIDs, or Global Entity IDentifiers. An Entity's GEID is it's position in the Entity Container plus the number of Entities on processes lower in rank than the current process. This number or address is static: no matter what operations you perform to an Entity (such as adding or removing Attributes or changing data values), the Entity's position in the Entity Container remains the same.

An Entity is really nothing more than a pointer to its local data. Groups of Entities that all have the same list of Clone Attributes (and hence the same amount of local data) are managed collectively by a Species object. Species is hidden from the user such that they should never need to access it directly. Essentially, it provides an array for local data and a list of pointers to the Attributes the Species represents. There is one entry in this local data array for every Entity that is managed by a given Species with enough raw data space to store the local data for that Entity and a pointer back to the Entity itself. This local data space does not define the Entity, so when an Entity has one of its Attributes added or removed, the local data space for that Entity is moved to the Species representing the new combination of Attributes and Entity's pointer to its data is updated, but the Entity itself doesn't move.

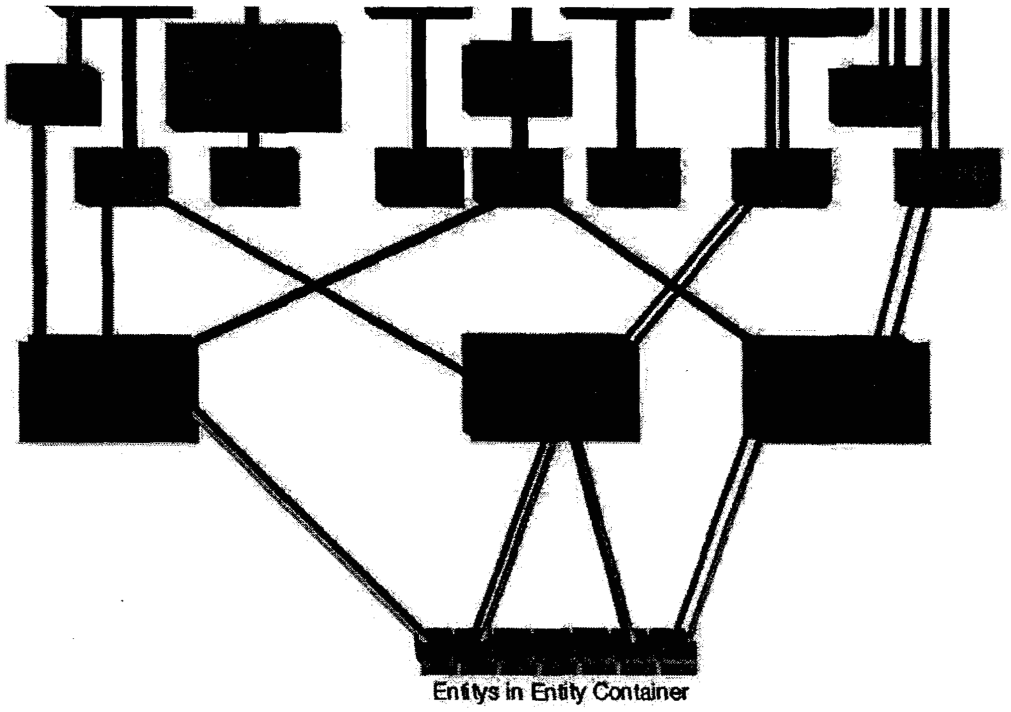


Figure 5: Entities being managed by Species.

Species keeps two lists of Attributes: the list of Attributes that it was created with as well as a list of Species Specific Attribute Copies. The Species Specific Attribute keeps some information on how to access its Local Data for Entities of that Species, specifically an offset into the local data space. The Species creates these Attribute Copies during its creation. If an Attribute has no Local Data, and hence, no need to store an offset in order to access all of its Data, the Species will just a pointer to the Clone Attribute that it asked for a copy of.

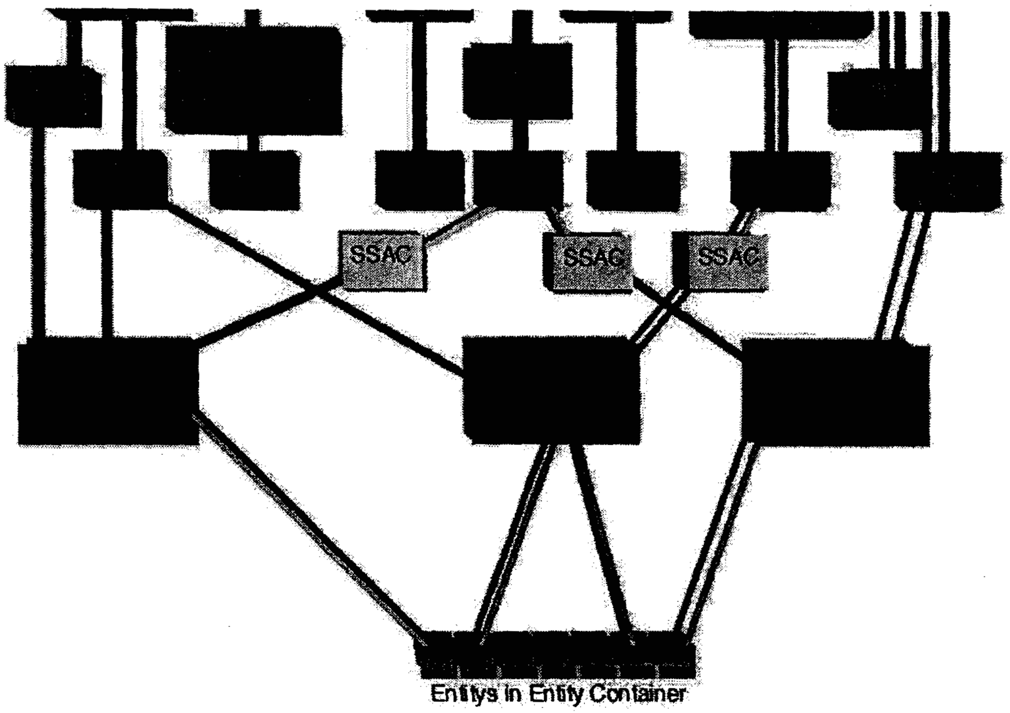


Figure 6: Entity-Attribute Structure with Species Specific Attribute Copies.

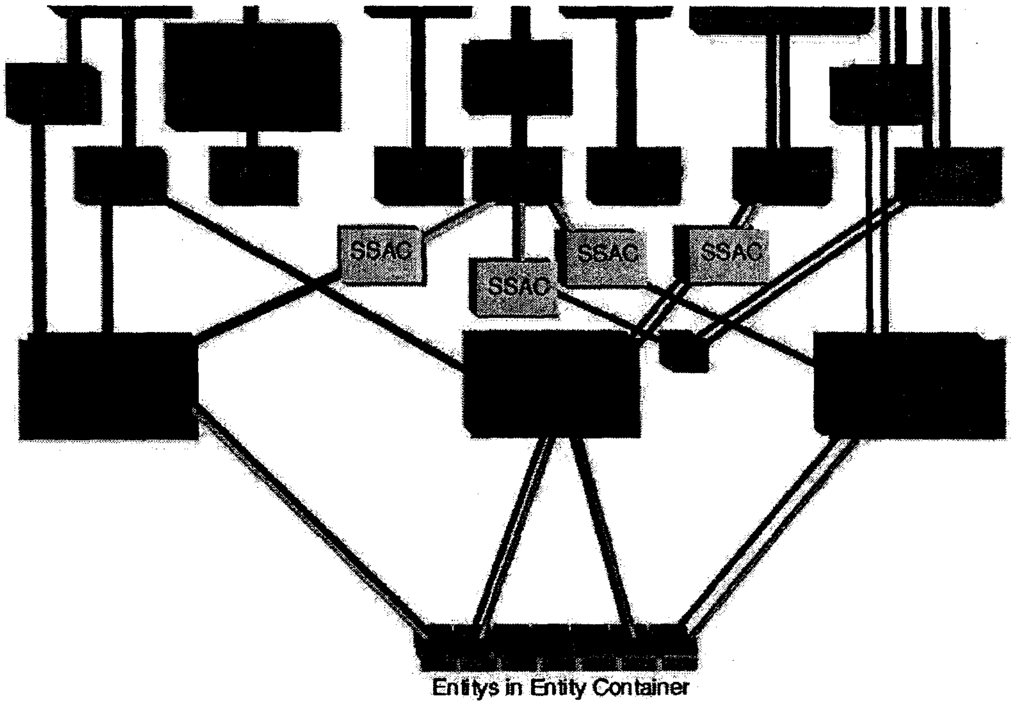


Figure 7: Entity-Attribute Structure after an Entity has changed one of its Attributes.

As it is the Attribute that contains the knowledge of how to access the Local Data for a given Entity, it is that Attribute that is called to access any of the Data that it has defined. This is accomplished by first requesting the instance of a given Attribute for a specific Entity. The Entity will pass this request on to its Species which then looks up the Attribute by ID in a table and returns the pointer to the Species Specific Copy of that Attribute. This can be done for any Clone Attribute or Type Attribute in the system. If the Entity does not have the requested Attribute, the user will get a NULL pointer. Recall that if the Entity has multiple Clones of a Requested Attribute, it is indeterminate which Attribute will be returned; the user should ask for the list of those Attributes instead. Once the user has a pointer to the appropriate Shared Specific Attribute Copy, any of that Attribute's accessor functions may be called using the Entity as an argument. The data accessor function can either access the data using the Entity and the stored offset, pass the Entity to a user defined function, or ignore it and directly access the Common Data. The procedure in the previous paragraph can be very cumbersome, especially since there are numerous pointer de-references and casts involved. In order to spare the user from having to repeat such a sequence needlessly and with the high potential of creating syntactic errors in the code, the EA macro was developed. The EA macro takes two arguments: A pointer to an Entity and the name of the Attribute Type the user would like from that Entity. The macro expands to retrieve the correct Attribute pointer for that Entity and that Type Attribute and allows the user to directly call the data accessor function from there.

An Entity is only owned by one process, however any other process can hold a copy of that Entity. These Entities are called Ghost Entities. A Ghost Entity is retrieved from the process that owns it by passing that Entity's GEID to the get_entity function. Hence, the user must know the GEID of the Entity they want. This is typically accomplished by virtue of having a connectivity between Entities. The Topology of a Mesh is the de facto example of this connectivity. The get_entity function is another example of asynchronous communication in TIGER-III. When the get_entity function is called, it first checks if the requested Entity is already in this process. If it is not, then get_entity sends a request to the process that owns the requested Entity (which get_entity can determine because the GEIDs are assigned to processes in blocks. That process has a transporter thread that receives the request and returns the list of Clone Attribute IDs that the requested Entity has and the Local Data for that Entity. Right now the Local Data is just sent as a group of bytes, so it will only work across homogeneous architectures within a given mpirun (as heterogeneous architectures will likely have different sizes and alignment for the data as well as different byte orderings). This is not a significant limitation given the lack of heterogeneous low-latency computing clusters at this time. The process that owns the Entity will tag that Entity with an Exists_on Attribute to denote that the requesting process has a copy of that Entity. The Entity is received and unpacked by the requester. When the requester unpacks the Entity, the Entity will be tagged with the Ghost Attribute to denote that it is owned by another process.

Given the organization of Entities by Species, the implementation of the Filter class is relatively simple. First the Filter finds the Type Attributes and Clone Attributes that fulfill the user's request. The list of Species for each of those Attributes can be considered to be a set. Depending on the operators specified in the filter string, the Filter will take unions or intersections of the sets. The set of Species that remains after this operation contains the Entities that the user wants to iterate over. The actual iteration process hidden by the Filter is nothing more than a double loop. The outer loop over the set of Species and the inner loop over the Entities of the current Species. The primary limitation in this approach is that Entities can not be filtered based on their Local Data values. This could be corrected through the use of a Visitor class that uses a Filter to get a group of Entities which it can then iterate over, returning only the Entities that the user has requested.

The Data for the Entities and Attributes should be the same in all processes. In order to do this, when a Common or Computed Data is updated for a given Clone Attributes (or one of its Species Specific Copies), a message is sent to the mass mutator (so named as it is essentially changing the values of Attributes for all the Entities with that Attribute) thread on a designated process (in order to create the same serialism as happens in the Attribute cloning process) which dispatches the message to all other processes. The message specifies the Clone Attribute to be updated, which Data is being updated and the new Common Data value or Computed Data function. Each process will find the given Data in the given Clone Attribute and pass it the new value or function. That Data will get the new value or function for the Clone Attribute and all of its Species Specific Copies.

Updating Local Data is slightly different because Common and Computed Data should always be updated in all processes, but Local Data should only be updated when the

Entity for that Local Data is a Ghost Entity in some process. Because all Entities of a given Species have the same Attributes, all of them will have the same Exists_on or Ghost Attributes. So a flag is stored in the copies of Data that a Species Specific Attribute Copy has, indicating whether or not the Entities of that Species need their Local Data communicated to some other processes when it is updated. If this flag is not set during a Local Data update, the new data value is pounded into the space designated by the offset into that Entity's local data space. If the flag is set, the Exists_on or Ghost Attribute for that Entity is found to determine what process owns that Entity. A message similar to the one described for Common and Computed Data is created with the addition of the GEID of the Entity in question. This message is sent to a mutator thread in the process that owns the Entity (again, an imposed serialism is created). That process will find the given Entity, see what processes it has an Exists_on Attribute for, and propagate the message to those processes. The data value is then updated on all processes with that Entity simultaneously.

The mutator thread can handle more than just updating Local Data. It also handles the removal, addition or setting of Attributes to or from an individual Entity, or the removal of an Entity altogether. The process is virtually identical to the process of updating Local Data. When a user requests one of these operations and the Entity exists in another process, a message containing the operation type, the GEID and the data needed for that operation (such as the list of Attribute IDs to add to the Entity) is sent to the local mutator thread. The mutator thread propagates the information out to all processes containing a copy of that Entity and the operation is performed by all processes simultaneously.

Mesh Classes

The Mesh class has been the driving factor behind the development of the Entity-Attribute structure. The Mesh class provides an abstraction for any Cell, Face, Edge, and Node type topology that a physics code may want to use. The mesh is initialized with a file that lists the mesh parts (by filename), and the type of mesh that is in the part file (unstructured, structured, warped, etc.). These are the meshes that will be stitched together and decomposed across processors, essentially defining the space in which the physics code will do its calculations. The Mesh class provides an interface such that the physics code does not need to be concerned with what type of mesh its working on, or what the partitioning of that mesh across all the processors of a massively parallel machine is.

A great deal of the setup for use of the meshes in this manner is performed by the TIGER III pre-processor (PreTiger) and the Mesh Readers. PreTiger only needs to be run once for a given collection of mesh parts. It will read in the mesh parts using the Mesh Readers, partition the parts across the processors that it is running on, and write out binary files to be used by the TIGER classes when the physics code is run. The primary limitation that it has is that it will only partition the mesh for the number of processors that it is running on, hence the physics code must be run on the same number of processors. If the number of processors for a run is changed, then PreTiger must be re-run to re-partition the mesh parts for the new number of processors.

PreTiger begins by creating a Mesh Reader for every part in the mesh parts file. The Mesh Readers are a hierarchy of classes to handle the reading of ASCII or binary mesh part files at various levels of abstraction. The root Mesh Reader is a virtual class that only defines those members that are applicable to all mesh part files, such as a function to retrieve the number of zones. Various types of abstract Mesh Type Readers, such as Warped and Unstructured inherit from Mesh Reader to define the interface for the instantiation of specific types of Mesh Readers, such as unstructured or warped. Using these readers, PreTiger determines the size of each part (in number of zones) and factors that with a weight depending on the type of mesh it is (i.e. Structured meshes have a smaller memory footprint than unstructured, so more Structured zones can be fit on a given processor).

Given the relative size information for each part, the parts are assigned processors: one part may be spread across multiple processors and one processor may be assigned multiple parts. The algorithm is designed to balance the work between processors without splitting a part between so many processors as to unnecessarily increase the communication throughout the system. Each processor will then process the meshes to which it was assigned. If it is the only processor assigned to that mesh, the mesh is read in and dumped out in binary format (for efficient reading when the physics code is started). One binary file is created for structured meshes, three binary files are created for unstructured meshes (one for nodes, one for cells and one for special surfaces).

If a structured mesh needs to be partitioned, a block partitioner is called. The block partitioner algorithm recursively breaks the problem down into smaller blocks until the number of blocks is equal to the number of processors assigned to that part. The blocks remain roughly the same size and retain a near-optimal area to surface ratio. Tests have shown that the block partitioner can break the problem to within 2% of the optimal partitioning on a moderate sized part of odd dimension across an odd number of processors. In general, the load imbalance decreases as problem size increases.

If an unstructured mesh is to be partitioned across multiple processors, those processors create a MPI Communicator [] to talk amongst themselves. Each processor reads in a section of the cells from the part file and the processors use the nodes in common between cells to determine the basic connectivity of the cells. This information is put into a graph form and passed to ParMetis[], which performs the domain decomposition on the part and assigns what processors work on what cells. The cells are then sent to those processors. The nodes and specials are read from the part file, again with each processor reading just a piece of the file. Each processor then requests the nodes (and specials associated with those nodes) from the processor that read them in, and the nodes returned to the processors that requested them. Finally, the nodes, cells and special surfaces that have been assigned to a given processor can all be written out to disk in their separate binary files. This entire procedure is obviously an over-simplification. There is a lot more book keeping involved to ascertain who read in what and where it currently is.

Furthermore, the entire procedure is multi-threaded for efficiency, for instance a processor will request the nodes that it needs at the same time as the cells are being written to their own binary file.

The partitioning of unstructured meshes is a late added feature to TIGER-III and, while it is complete, it is not yet functional due to problems with MPI communication. The foremost problem that has been encountered is with MPI Communicators in the SGI IRIX MPI implementation. PreTiger depends on the processes in a communicator to be numbered from 0 to n-1 where n is the number of processes in that communicator[]. Instead, on the SGI they retain the same rank they have in MPI_COMM_WORLD. Hence, if processes 2 and 4 are the two processes in a communicator, PreTiger expects them to be numbered 0 and 1 for that communicator. Instead, the processes remain numbered 2 and 4.

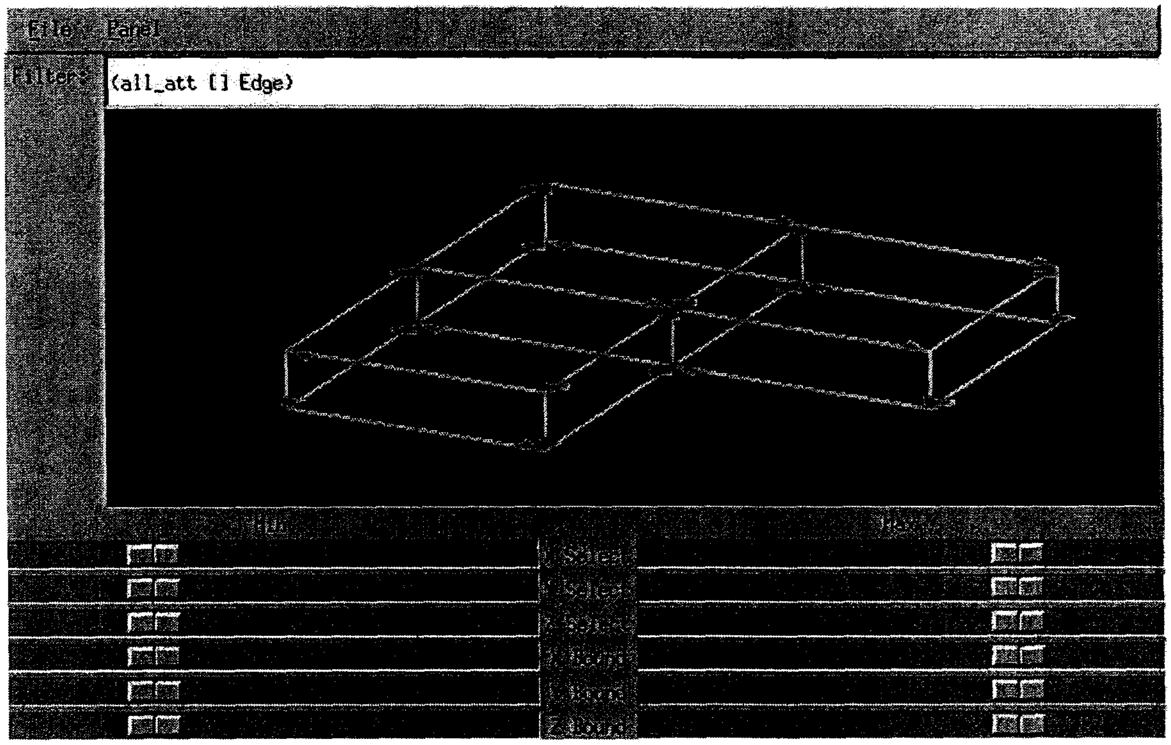
Once the mesh is partitioned and saved in a binary file format, the actual physics code based on the Mesh class can be executed. The same parts file used for PreTiger is used to initialize the mesh class. Each process sees if there's a binary file for that part created for that process. If there's not, it can go on to the next part. If there is, then the appropriate mesh part (unstructured, warped, etc.) is constructed.

Each node, edge, face and cell is represented by a Mesh Entity, which is a specialization of Entity that understands mesh topology. A Mesh Entity is an Entity with the Topology Attribute. The Topology Attribute defines the number of ups or downs a Mesh Entity has, as well as provides a place to store those ups or downs for unstructured meshes (the ups and downs are Calculated Data in structured meshes, hence the memory savings). The ups and downs for unstructured meshes are stored as Local Arrays. Local Arrays can be tricky to deal with, particularly when an item (such as an up or down) needs to be added or removed. This is the primary motivation behind the Mesh Entity class, as it encapsulates all this complexity from the rest of the system. All of the Mesh Entities for all parts in the system are stored in a single Entity Container, which knows how to manage them collectively.

All of the structured parts need to be read in and constructed first. Structured parts (both basic and warped) are defined by their size and origin. These are stored in a Cfen_block (Cell Face Edge Node Block Cfen_block) object which handles all the calculations for moving through the topology of a given structured block. The Cfen_block specifies how many Mesh Entities are required to store that part, and a block of that many Mesh Entities is allocated from the Entity Container. Most of these Entities have no overhead other than the basic two pointers: all of their Data is Common or Computed. The main exception is the local space to store the positions of nodes in a warped mesh. The second to outside layer is pseudo-unstructured, that is, it's labeled as being unstructured and it stores the ups and downs locally. This is because the Mesh Entities on the outside layer might already exist (for instance if two structured mesh parts connect). In this case, the Mesh Entity isn't duplicated to maintain the structured ordering, rather the existing Mesh Entity is used and the second to outside layer's ups or downs will point to it.

It can easily be determined if another node already exists in our problem space by placing all of the nodes in an oct-tree. Each node is inserted into the oct-tree by position. Then, before a new node is created, the oct-tree can be checked to see if that node already exists. If it does, then the existing node is used. This allows meshes to be easily stitched together such that all the mesh parts assemble into one global topology. Each node stored in the oct-tree is given a tag. If the tags do not match, the nodes will be multiply inserted. Most tags will be a default value, however coincident nodes, such as will appear on slide surfaces or special electromagnetic boundary condition surfaces can use different tags to create a logical boundary to the problem at that location.

Unstructured mesh parts are created by reading in all the nodes for that part, and creating a Mesh Entity for each one that doesn't already exist. Then the cells are read in, a Mesh Entity is created for each one, and all the nodes for that cell are located. Any edges or faces needed to describe that cell that don't already exist are created and the nodes, edges, faces and cell are connected together through the addition of the respective ups and downs.



Filter: (all_att [] Edge)

Figure 7: Two mesh parts (one with two cells, one with one cell) loaded in together and treated as a single mesh.

Once all the mesh parts are read in and constructed, this process has all the Mesh Entities for which it handles the physics. However, physics doesn't conveniently stop at the process boundaries. The user needs to be able to access and update Mesh Entities that are topologically adjacent to the ones controlled by this process. Hence, once local mesh construction is complete, global mesh construction takes place. First, the global problem space is broken up into sectors. Each process enumerates what sectors it has nodes in though the use of the oct-tree. This list is sent to a designated process (usually 0) which returns a list of the processes this process shares sectors with. This is the neighbor list for this process, to first order. The outer layer of nodes is sent and received from every process in that neighbor list. Ownership is determined for each node that is shared with another process. Specifically, the process with the highest rank that has a node owns it. The node is tagged as a Ghost node in all other processes. This means that the process that is highest in rank will own its entire outer surface, whereas process 0 will only own any part of the outer surface that forms the outside boundary to the problem. The ownership of edges and faces on the outside surface can be similarly determined, however those Mesh Entities only need to be sent to the processes that have one of the nodes they connect to (as determined in the previous step). Finally, the ups, downs and any special Attributes are consolidated in the process that owns each Mesh Entity on the outer surface. This information is passed to the processes that store those Mesh Entities as ghosts.

The ups and downs are usually stored as Mesh Entity pointers. This presents a problem if a Mesh Entity in this process refers to Mesh Entities of which there are not copies in this process. In that case, the GEID for the Mesh Entity is stored instead. This creates a new problem: there may not be a discernible difference between GEIDs and Mesh Entity pointers. Hence, instead of mixing the two, a Mesh Entity either has Mesh Entity pointers for all of its ups and downs, or it has GEIDs for all of its ups and downs. The former is either a Mesh Entity that is owned in this process, or a Ghost Mesh Entity. The latter is referred to as a Phantom Mesh Entity (as in a Mesh Entity that's not really there). The first time a phantom's ups or downs are accessed, all of its GEIDs will be converted into pointers and it will become a normal Ghost Mesh Entity. The resulting mesh in any given process will be surrounded by a layer of Ghost Mesh Entities of arbitrary thickness. The outer most layer stored in any process will consist of Phantom Mesh Entities. This excludes the true outer boundary of the problem, of course. The main advantage of this approach is that it allows the physics code to access zones owned by other processes arbitrarily. Some codes only require a single layer of ghost zones. Others have complicated stencils that may go five ghost zones deep in places. Regardless, the user has been relieved of the burden of finding, loading and managing those ghost zones.

Summary

Our Experiences using C++

In summary, we have learned much during this investigation. There are many challenges to using object-oriented software techniques in scientific settings. While we believe the benefits still far outweigh the limitations. The following three paragraphs describe some general limitations we have found using C++ on the project. The comments are specifically directed to our experiences on this project trying to use C++ for a specialized high-performance scientific applications.

We spent too much time wrestling with C++ compiler deficiencies.

- lack of ANSI standard C++ implementations (This has improved significantly over the last 3 years), especially full template support, member template functions

We wrestled too much with C++ syntax and performance issues. Many times throughout the project we would come up with a potentially acceptable design for a given set of classes but the implementation was very difficult. Often the most elegant solution was too liberal in its use of virtual pointers. However, virtual pointers are expensive both in memory and CPU usage. The overhead of a virtual pointer is not limited to the virtual call itself but also limits the compiler's ability to inline and optimize. First implementations could be 5 times slower and use significantly more memory. These inefficiencies by themselves might be acceptable but typically grew slowly to unacceptable levels.

A good deal effort was expended on the project to trying to make Entities abstract. Having Flyweight [Gamma, et. al., p. 195] objects is very general and allows a great deal for a great flexibility. Moving the abstractions up to deal with collections of items and the management of containers reduces the abstraction performance penalty significantly. TIGER-I implemented a series of container classes and management classes that manipulated data structures consisting of large sets of raw data. TIGER-II and TIGER-III added the capability of abstracting at a lower level. Every Node, Edge, Face, Cell in the entire program could be a real object. This approach gave us a great deal of object-oriented freedom but also significantly increased the number of performance and implementation issues we had to address.

One of the ways we were able to improve efficiency was by doing much of the objected-oriented setup during the initialization phase. This helped performance but forced many of our setup operations to occur before main() was called. Several man-months on the project were expended trying to enforce a proper static initialization of global objects across multiple file units. We ended up having to develop a modified Singleton design pattern [Gamma, et. al., p. 127] where we wrapped the initialization of static data within a member functions. We still need to be very careful in our linking phase when we create a library. Our solution appears to work for on native SGI and DEC-Alpha clusters compilers, and g++ and KCC (Kuck and Associates) on the SGI, DEC, and ASCI Blue (IBM SP-2) platforms. (latest current versions).

In retrospect:

We had 3 major design cycles in the project. At the end of TIGER-1 we were very happy with the accomplishments. We were able to test our abstractions out by building full working code capable of modeling a kicker structure beyond what could be done prior to the project. It was our prototype version and we knew we could do even better with our next iteration. We were also pleased with the progress we were making during the development of the TIGER-II version. In fact, originally TIGER-II was to be our fully developed final version. However, as we added more and more capabilities the complexity of our implementation increased significantly. First we added unstructured grids, then the capability to have multiple unstructured grids, then the ability to have structured grids, then the capability to automatically stitch arbitrary structured and unstructured grids together, then the ability to have extruded meshes, then the ability to have revolved meshes, and finally the ability to have any number of any structured, warped, extruded, revolved, or unstructured meshes together. The user interface was still very clean and the bookkeeping details were hidden behind the interface. The programmer was successfully insulated from many of the bookkeeping details we were trying to abstract away. During the development of TIGER-II we also began to test the massively parallel abstractions out. TIGER-II's parallel implementation was only partially completed. It worked for unstructured but needed to be extended for hybrid meshes. TIGER-II's parallel implementation also relied on a serial pre-processor that read in the mesh on a single processor and then called the mesh partitioner.

During the second quarter of FY99, almost 2.5 years into the project we came to a decision point. We were achieving our milestones and were happy with our successes. However, although we successfully implemented hybrid mesh abstractions that provided a simple interface to the programmer, hiding virtually all the details, the complexity behind the interface raised concerns about the long term maintainability, flexibility, and performance of our libraries. On one hand, future physics packages would be spared much of this complexity, and after all, the insulation of the physics from the bookkeeping was what we were trying to achieve. There would hopefully be many physics packages but we only needed one bookkeeping package and this was essentially written. On the other hand, through our research we discovered ways to significantly reduce the complexity of our implementation and ways to further improve the memory overhead and flexibility of our interface.

A major redesign and implementation of our abstractions would be a huge undertaking so late in the project even if we used the latest in modern software techniques. However, the expected increase in capability, flexibility, maintainability, and performance, as well as the expected decrease in complexity would dramatically improve this research effort. It was a very tough decision to make.

There was one more factor influenced the decision making process. At the same time we were also trying to incorporate new physics into the full working TIGER-I version of the code. The new physics was far from trivial. We were trying to add correction terms to sources to the DSI algorithm. The correction terms were required to preserve divergence and prevent change build-ups that often lead to instabilities. In very simple terms, the solution to adding the correction terms involved turning the DSI algorithm inside out. This was a good test for the Mesh abstractions. Previously we tested the abstractions knowing exactly what physics we were going to add next. This test would in some sense test our ability to add new physics. The addition of the current sources was not part of this LDRD project but a Techbase project. The FY99 Techbase report documents that effort []. What is relevant to this project is the lessons we learned testing out our prototype abstractions in a realistic setting.

We successfully added the new physics much faster than would have previously been possible, modifying the original FORTRAN version; yet we desired the process to be even simpler. We had long since improved the interface and hid many more of the details in TIGER-II version compared to the earlier TIGER-I version. However, the overall process was still dominated by bookkeeping details. The actual physics addition was relatively straightforward. The difficulty was that the algorithm forced us to iterate through the mesh in a very indirect manner. So during the physics addition, we had to get our hands dirty and go into the lower level bookkeeping aspects of the code. This is exactly what we were trying to prevent. The experience raised several questions. Was this example a typical or atypical situation? How often would we truly be successful in insulating the physics for the bookkeeping?

The decision was made to build a third version called TIGER-III. Having a third version start so late in the project meant that we would not have the time to put new physics into the last version to demonstrate its capabilities. We felt however that the long-term future of our research would be better served by building a better set of abstractions rather than finishing and demonstrating the TIGER-II version.

TIGER-III is in its final stages and has no physics in it yet. However, we have successfully moved beyond many of the difficulties encountered in earlier versions. It is our intention to finish and then demonstrate the object-oriented research behind TIGER-III in future publications. TIGER-III's core libraries are currently being used in another Mesh generation LDRD project.

[1] Design Patterns: Elements of Reusable Object-Oriented Software; Gamma, Helm, Johnson & Vlissides; Addison Wesley, 1994, pp. 1-395.

zow@congo.llnl.go

SPARCprinter

Tue Feb 15 14:14:32 2000

Utility Classes
Appendex A

NeWSprint 2.5

Needs .
Appendix covers yet.

~~Disclaimer~~

This is not source code.

There are no constants like "speed of light" etc... that I am aware of

David Steich

Contents

1 What's in Array.H?

Names

These array classes provide us with more safety and flexibility than the standard array operators. Safety is provided through such means as range checking and flexibilty from adjustable arrays. There is negligable speed loss compared with C arrays. We rely on templates heavily due to performance concerns. At present there are no virtual member functions except for the destructors.

Currently the best performance is achieved by inlining a vast majority of the Array member functions. Even larger sized member functions are sometimes inlined if they are:

1) very heavily used,

2) not inlined at many places,

3) tested (in a very limited sense) to improve performance.

Obviously testing performance is compilier, system, and version dependent. It is expected that this will change with time. Unfortunately, overall application performance can still change by several hundred percent by changing inlines of the Array member functions.

Maybe we could look at inlining and performance in a systematic way in the future. For now we are just doing simple, limited in scope sanity checks.

1.1

extern Monitor **stack_monitor**

Provides a monitor to all Stack_Arrays, regardless of their templated type

Provides a monitor to all Stack_Arrays, regardless of their templated type. This has been done to minimize Stack_Array's memory useage. Global variables increase program complexity, and therefore should be avoided, however this one is necessesary to keep a reasonable memory footprint.

1.2

template < class Type, Index SIZE> class **Stack_Array**

The stack array class provides an array of the given type and size to its children

Public Members

Stack_Array (Index)
The constructor allows us to track stack memory useage.

˜Stack_Array () *The destructor allows us to track stack memory useage.*

Monitor* **monitor** () *Get a monitor to protect array when multiple threads are in use*

1.2.1 inline Type&
operator() (const Index index)
Get a reference to the element at a given position 8

1.2.2 inline Type&
ref (const Index index)
Get a reference, as in operator() (so this needs to be protected in the same way), but fast like operator[] (no resizing) 8

1.2.3 inline const Type&
operator[] (const Index index) const
Find the element at the given index, just like a standard C array

9

inline void
update (const Index position, const Type& item)
Update the element at the given position to the given value.

inline void

		insert (const Index position, const Type& item)	*Update the element at the given position to the given value.*	
1.2.4	const Type&	**data** (const Index i) const	*Provides uniform interface for array classes to access data_*	9
1.2.5	Type&	**data** (const Index i)	*Provides uniform interface for array classes to access data_*	9
1.2.6	const Type*	**data** () const	....	9
1.2.7	Type*	**data** ()	....	10
	Index	**capacity** () const	*Return the capacity of this array.*	
	Index	**capacity** (Index) const	*Return the capacity of this array regardless of index*	
	Index	**size** () const	*Return the size of this array.*	
	Index	**size** (Index)	*Return the size of this array regardless of index*	
	void	**sort** ()	*Sort this array.*	
	Index	**find** (const Type& item) const	*This function is guaranteed to find the item you are looking for, or it will die, so only call it when you know the item is in the array*	
1.2.8	Index	**find_position** (const Type& item) const	*Finds the position that the given item is at or should be at*	10
	Index	**index** (Type * address) const	*Find the position in the array given the address.*	
	template <class Rhs_type> inline void	**equal** (const Rhs_type& rhs)	*Essentially a templatized operator= (NOT ==) used by operator= in Container*	
	inline Stack_Array <Type, SIZE> &			

operator= (const Stack_Array<Type, SIZE>& sa)
Equals operator. For completness and to remove compilier warnings

inline void
relocate (Stack_Array<Type, SIZE>*)
Classed when a relocation of array has occurred.

Protected Members

1.2.9 void **set_size** (const Index sz)
Set the size of array 10

The stack array class provides an array of the given type and size to its children. This data will be on the stack provided it isn't too large and is part of a static or automatic (not new'ed) object.

1.2.1

inline Type& **operator()** (const Index index)

Get a reference to the element at a given position

Get a reference to the element at a given position. Will increase the size and capacity of the array as necessasary. Not thread-safe - be sure to increment the monitor before use (and decrease it when you're done).

1.2.2

inline Type& **ref** (const Index index)

Get a reference, as in operator() (so this needs to be protected in the same way), but fast like operator[] (no resizing)

Get a reference, as in operator() (so this needs to be protected in the same way), but fast like operator[] (no resizing). Not that any of that matters for Stack Array.

1.2.3

inline const Type& **operator[]** (const Index index) const

Find the element at the given index, just like a standard C array

Return Value: s A constant reference to the element.

1.2.4

const Type& **data** (const Index i) const

*Provides uniform interface for array classes to access data_*

Provides uniform interface for array classes to access data_. Not thread safe: increment monitor before use.

1.2.5

Type& **data** (const Index i)

*Provides uniform interface for array classes to access data_*

Provides uniform interface for array classes to access data_. Not thread safe: increment monitor before use.

1.2.6

const Type* **data** () const

Return Value: s a pointer to beginning of valid data_ address space. Not thead safe.

1.2.7

Type* **data** ()

Return Value: s a pointer to beginning of valid data_ address space. Not thead safe.

1.2.8

Index **find_position** (const Type& item) const

Finds the position that the given item is at or should be at

Finds the position that the given item is at or should be at. It will die if the item is not in the array and there is not room for it; however, as Stack_Arrays have no concept of size vs. capacity, it acts the same as find() .

1.2.9

void **set_size** (const Index sz)

Set the size of array

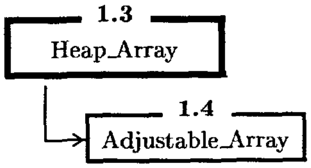
Set the size of array. This member function is provided only to allow for a uniformity among other types of arrays. An assert failure occurs if user actually trys to change size of an Stack array.

1.3

template <class Type> class **Heap_Array**

An array located in the heap

Inheritance



Public Members

1.3.3		**Heap_Array** ()	*Default constructor*	14
		Heap_Array (Index n)	*Construct a heap array of the given size.*	
		Heap_Array (const Heap_Array<Type> &s)	*Copy constructor.*	
	Monitor*	**monitor** ()	*Get a monitor to protect array when multiple threads are in use*	
	Index	**capacity** () const	*Get the array's capacity*	
	Index	**capacity** (Index)	*Get the array's capacity*	
	Index	**size** () const	*Get the array's size*	
1.3.4	inline Index	**size** (const Index sz)	*Set the number of used elements in this Array*	14
1.3.5	inline Type&	**operator()** (const Index index)	*Get a reference to the element at a given posistion*	14
	inline Type&	**ref** (const Index index)	*Get a reference, as in operator() (so this needs to be protected in the same way), but fast like operator[] (no resizing)*	
1.3.6	inline const Type&			

operator[] (const Index index) const
Find the element at the given index, just like a standard C array
15

inline void
update (const Index position, const Type& item)
Update the element at the given position to the given value.

1.3.7 inline void
insert (const Index position, const Type& item)
Insert the given value at the given position 15

Index **find** (const Type& item) const
This function is guaranteed to find the item you are looking for, or it will die, so only call it when you know the item is in the array

1.3.8 Index **find_position** (const Type& item) const
Finds the position that the given item is at or should be at 15

1.3.9 Index **find_ordered_position** (const Type& item) const
Finds the ordered position an item should be located at if it were inserted in array 16

void **sort** () *Sort this array.*

Index **index** (Type * address) const
Find the position in the array given the address.

template <class Rhs_type> inline void
equal (const Rhs_type& rhs)
Essentially a templatized operator= (NOT ==) used by operator= in Container

inline Heap_Array <Type> &
operator= (const Heap_Array<Type>& ha)
Equals operator. For completness and to remove compilier warnings

inline void

relocate (Heap_Array<Type>*)
Called when a relocation of array has occurred.

~**Heap_Array** () *Destructor*

Protected Members

Type* **data_** *Pointer to beginning of actual data*

Index **size_** *The size of the array the user presently has access to*

Index **capacity_** *The actual memory capacity this array owns*

Monitor **monitor_** *Monitor for threading*

inline void
set_size (const Index sz)
Set the size of the array (capacity will change only if needed)

Type* **data** () *Return handle to data*

const Type*
data () const *Return const handle to data*

Type& **data** (const Index i)
Return handle to data position i

const Type&
data (const Index i) const
Return const handle to data position i

1.3.1 inline Type*
new_memory (const size_t n_ele)
Gets memory from the memory manager and calls default constructor for each newed element
16

1.3.2 inline void
delete_memory (Type *old, const size_t n_ele)
Release memory back to memory manager calling default destructor for each deleted element 16

An array located in the heap. Also known as Fixed Array as the capacity is fixed. By default the size is set to be the same as the capacity, but the user can set it to be anything from 0 -> capacity.

1.3.3

Heap_Array ()

Default constructor

Default constructor. Intentionally does nothing overriding the compilier generated default constructor.

1.3.4

inline Index **size** (const Index sz)

Set the number of used elements in this Array

Set the number of used elements in this Array. Heap_Array (F_a) is fixed in capacity, but the size is adjustable.

1.3.5

inline Type& **operator()** (const Index index)

Get a reference to the element at a given posistion

Get a reference to the element at a given posistion. Will increase the size of the array as necessasary. Not thread-safe. Be sure to incriment monitor before use and decriment it when you're done.

1.3.6

inline const Type& **operator[]** (const Index index) const

Find the element at the given index, just like a standard C array

Return Value: s A constant reference to the element.

1.3.7

inline void **insert** (const Index position, const Type& item)

Insert the given value at the given position

Insert the given value at the given position. This will increase the size of the array if necessary.

1.3.8

Index **find_position** (const Type& item) const

Finds the position that the given item is at or should be at

Finds the position that the given item is at or should be at. It will die if the item is not in the array and there is not room for it.

1.3.9

Index **find_ordered_position** (const Type& item) const

Finds the ordered position an item should be located at if it were inserted in array

Finds the ordered position an item should be located at if it were inserted in array. The search is sequential from the beginning of the array. Note that the search item will return the last valid position where all items are <= to the item.

1.3.1

inline Type* **new_memory** (const size_t n_ele)

Gets memory from the memory manager and calls default constructor for each newed element

Gets memory from the memory manager and calls default constructor for each newed element. Unless overloaded, this member function provides a single place where memory for derived classes is retrieved by the memory manager. Changes to this member function must be tied to changes in delete_memory member function.

1.3.2

inline void **delete_memory** (Type *old, const size_t n_ele)

Release memory back to memory manager calling default destructor for each deleted element

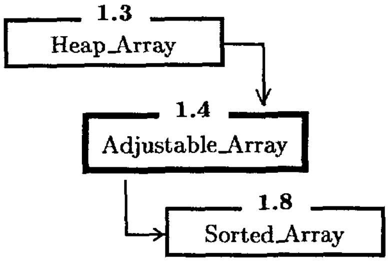
Release memory back to memory manager calling default destructor for each deleted element. Unless overloaded, this member function provides a single place where memory for derived classes is released back to the memory manager. Changes to this member function must be tied to changes in new_memory member function.

1.4

template <class Type> class **Adjustable_Array** : public Heap_Array<Type>

An extention of the Heap_Array that allows the size and capacity to be dynamically adjusted

Inheritance



Public Members

Adjustable_Array (const Index n)
Construct an adjustable array of the given size

Adjustable_Array (const Adjustable_Array<Type> &s)
Copy constructor.

1.4.3 inline Type& operator
() (const Index i) *Not thread-safe* 19

1.4.4 inline void
insert (const Index i, const Type& item)
Insert the given value at the given position 20

1.4.5 inline void
insert (const Index i, const Type& item,
const Index new_capacity)
Set the capacity of the array to the given value and insert the given value at the given position 20

1.4.6 inline void

insert (const Type& item)
Insert the given item in the last position in the array, increasing the size of the array by one 20

inline void
last (const Type& item)
Insert the given item in the last position in the array, increasing the size of the array by one

inline Index
size (const Index n)
Set the size of the array to the given value.

Index **size** () const *Get the size of the array.*

inline Index
capacity (const Index new_capacity)
Set the capacity of the array to the given value.

Index **capacity** () const *Get the array's capacity*

template <class Rhs_type> inline void
equal (const Rhs_type& rhs)
Essentially a templatized operator= (NOT ==) used by operator= in Container

inline Type*
intersection (const Heap_Array<Type> &a)
Find the first intersection element between this array and a

inline void
intersection (const Heap_Array<Type> &a,
const Adjustable_Array<Type> &b)
Fill array b with the entire intersection of elements between this & a.

1.4.7 inline void
merge (const Heap_Array<Type> &a,
const Adjustable_Array<Type> &b)
Merge this array with a and place the union array in b 21

inline void

push (const Type&)
Push item on array in conventional stack fashion

inline void
pop () *Pop item on array in conventional stack fashion*

inline Type
top () *Take the last item off the array in conventional stack fashion*

inline Adjustable_Array <Type> &
operator= (const Adjustable_Array<Type>& aa)
Equals operator. For completness and to remove compilier warnings

1.4.8 **˜Adjustable_Array** ()
Default destructor 21

Protected Members

1.4.1 inline void
set_size (const Index n)
Sets the size of the array to the given value, increasing the capacity if necessasary 21

1.4.2 inline void
set_capacity (const Index new_capacity)
Sets the capacity of the array to the given value 22

1.4.3

inline Type& operator () (const Index i)

Not thread-safe

Not thread-safe. Use the monitor to protect yourself by locking it down, as it may call set_capacity.

1.4.4

inline void **insert** (const Index i, const Type& item)

Insert the given value at the given position

Insert the given value at the given position. This will increase the size and capacity of the array if necessasary.

1.4.5

inline void **insert** (const Index i, const Type& item, const Index new_capacity)

Set the capacity of the array to the given value and insert the given value at the given position

Set the capacity of the array to the given value and insert the given value at the given position. This will increase the size and capacity of the array if necessasary.

1.4.6

inline void **insert** (const Type& item)

Insert the given item in the last position in the array, increasing the size of the array by one

Insert the given item in the last position in the array, increasing the size of the array by one. This member function is eqivalent to last() and is provided for seamless usage Adjustable and Sorted Arrays. Note that although syntax is the same between Adjustable and Sorted Arrays the behavior is markedly different. This routine will put the item in the list regardless of whether it already exists.

1.4.7

inline void **merge** (const Heap_Array<Type> &a, const Adjustable_Array<Type> &b)

Merge this array with a and place the union array in b

Merge this array with a and place the union array in b. Note b is an Adjustable array

1.4.8

~Adjustable_Array ()

Default destructor

Default destructor. Deliberately not virtual. Does nothing, Heap_Array's destructor does the work.

1.4.1

inline void **set_size** (const Index n)

Sets the size of the array to the given value, increasing the capacity if necessasary

Sets the size of the array to the given value, increasing the capacity if necessasary. Not thread safe: designed to be called from a member function that already has the array locked down.

1.4.2

inline void **set_capacity** (const Index new_capacity)

Sets the capacity of the array to the given value

Sets the capacity of the array to the given value. Not thread safe: designed to be called from a member function that already has the array locked down.

1.5

template <class Hidden_Data> struct **Block_header**

Block_header is a struct that contains data at top of every block in a Reference_Nc_Array

Members

	void*	**reference_**	*reference tag used by the Reference_Nc_Array user MUST be first item in the header so mod to address operations work*	
	Index	**running_index**	*running index of how many elements preceed this block*	
	Hidden_Data	**user_data_**	*User's hidden data that is packed into top of every block*	
1.5.1	long double	**first_data**	*We don't know at compile time what the alignment of the data is so we will use a safe long double*	23
		Block_header ()	*Default constructor*	
		Block_header (const Block_header<Hidden_Data>& bh)	*Copy constructor. For completness and to remove compilier warnings*	
	size_t	**header_offset** () const		

Return the required size in btyes of the block header data

inline Block_header <Hidden_Data> &
operator= (const Block_header<Hidden_Data>& rhs)

Equals operator. For completness and to remove compilier warnings

Block_header is a struct that contains data at top of every block in a Reference_Nc_Array. The primary reason for this struct is to let the system determine the address of first_data. This alleviates a fair amount of pointer arithmetic. The Block header stores a reference pointer guaranted to be the first data item so that address masking can be used for quick access. The header also stores a running index of how many items proceed this block an an area for user data. This could be a nested class but alas some compiliers are still behind the times.

1.5.1

long double **first_data**

We don't know at compile time what the alignment of the data is so we will use a safe long double

We don't know at compile time what the alignment of the data is so we will use a safe long double. first_data is not really used other that to take its address for alignment calculations.

1.6

template <class Type, class Hidden_Data> class **Block**

The Block class primary purpose is to encapulate a Block_header and its associated data

Public Members

Block () *Default constructor*

1.6.1 **Block** (const Block<Type, Hidden_Data>& blk)
Copy constructor 24

inline void
initialize (void* ref, const Index run_i,
const Hidden_Data& hd)
Inititialize the block

inline void
reference (void *ref)
Set the reference at the top of the block

inline void*
reference () const *Return the reference at top of the block*

inline char*
begin_data () *Return the pointer to the beginning of valid data*

inline size_t
header_offset () const
Return the required offset in bytes before beginning of data

inline Type&
operator[] (const Index bytes) const
Return item at location bytes offset from top of block

The Block class primary purpose is to encapulate a Block_header and its associated data. This class should be nested inside Reference_Nc_Array but alas some compiliers are behind the times.

1.6.1

Block (const Block<Type, Hidden_Data>& blk)

Copy constructor

Copy constructor. For completness and to remove compilier warnings.

1.6.2

inline Block <Type, Hidden_Data> & **operator=** (const Block<Type, Hidden_Data>& rhs)

Equals operator

Equals operator. For completness and to remove compilier warnings.

1.7

template <class Type, class Hidden_Data> class **Reference_Nc_Array**

Reference_Nc_Array (R_a) is an of array of Blocks

Public Members

	Reference_Nc_Array (const Index n, const Index bs = 4*KILO, const Index jmp = 0, Type* init = 0)	*Construct a Reference_Nc_Array of n elements each of jump size in blocks of bs with an optional initializer*
	Reference_Nc_Array (const Reference_Nc_Array<Type, Hidden_Data>& ra)	*Copy Constructor.*
size_t	**ref_mask** () const	*Return the number of bytes in each of the elements*
Index	**jump** () const	*Return the element jump size in bytes*
Index	**block_size** () const	*Return the number of elements that fit within a block*
Index	**num_blocks** () const	

Return the number of blocks in the array

inline Type&
operator[] (const Index i) const
Container's operator== wants a const version

1.7.2 inline Type&
operator) (const Index i)
Return reference to the i'th location resizing if necessary 28

Type& **ref** (const Index i) *Get a reference, as in operator() (so this needs to be protected in the same way), but fast like operator[] (no resizing)*

Monitor* **monitor** () *Return the threads monitor for this array.*

inline Type&
last () *Return the address of one past the end of the current size*

inline void
update (const Index i, const Type& item)
Set the array's i'th element to item

inline void
insert (const Index i, const Type& item)
Insert item at i'th location

inline Index
size (const Index sz)
Set the size of array changing capacity of necessary

Index **size** () const *Return the number of items*

inline Index
capacity (const Index new_capacity)
Set the new capacity

Index **capacity** () const *Return present capacity.*

inline void
sort () *Sort the array.*

inline void

reference (void * ref)
Return the reference guaranteed at the top of the block

void* **reference** () const *Return a reference to the "reference" pointer at top of each block*

inline Index
index (const Type* t) const
Find the position in the array given the address.

void **set_data_fx** (void (*fx)(Index, Hidden_Data*))
Pointer to function used to set user data at top of each block

inline Hidden_Data
hidden_data (void * pt)
Return a copy of the hidden data at top of block

template <class Rhs_type> inline void
equal (const Rhs_type& rhs)
Essentially a templatized operator= (NOT ==) used by operator= in Container

Reference_Nc_Array <Type, Hidden_Data> &
operator= (const Reference_Nc_Array<Type, Hidden_Data>& ra)
Equals operator. For completness and to remove compilier warnings

˜Reference_Nc_Array ()
Default destructor. Deliberately not virtual.

Reference_Nc_Array (R_a) is an of array of Blocks. Each Block contains a header. This header is memory aligned so that a reference can be found by taking any location inside the block (the address of an element) and masking off a given number of bits using the formula

element_address & ref_mask

where "&" is the bitwise AND operator. Each block also stores a running index of the number of elements in the array that are stored before this block. Each block also stores some user data. This data is set by a user provided function and retrived by giving any address in the block to the hidden_data member function.

1.7.2

inline Type& **operator)** (const Index i)

Return reference to the i'th location resizing if necessary

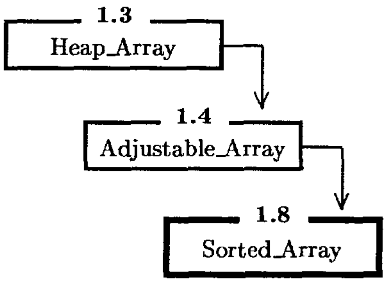
Return reference to the i'th location resizing if necessary. Not thread-safe. Use the monitor to protect yourself by locking it down as we may do set_ calls.

1.8

template <class Type> class **Sorted_Array** : public Adjustable_Array<Type>

An extention to Adjustable_Array that maintains the elements in sorted order resulting in log2(n) searches for a specific element

Inheritance



Public Members

1.8.1 Type& **operator()** (const Index) 30
inline Index

size (const Index n)
Ignore commands to set the size to anything larger than the current size

Index **size** () const *Get the size of the array.*

inline void
update (const Index, const Type&)
Sorted arrays don't allow direct access to elements

Index **insert** (const Index, const Type& item)
Ignore any index on an insert operation

1.8.2 inline Index
insert (const Type& item, const bool lock=true) ... 31

inline void
remove (const Type& item)
Removes the item or does nothing if the item wasn't found

inline void
remove_index (Index)
Removes the item at given index

1.8.3 inline Index
find_position (const Type& item,
bool * found) const
Finds the position where an item is or should be in the array 31

inline bool
has_item (const Type& item) const
Checks to see if item exists in array or not.

inline Index
find_position (const Type& item) const
Finds where an item is or should be in the array.

inline Index
find (const Type& item) const
Finds where an item is in the array. Dies if the item isn't found.

template <class Rhs_type> inline void

equal (const Rhs_type& rhs)
Essentially a templatized operator= (NOT ==) used by operator= in Container

inline void
equal (const Sorted_Array& rhs)
Essentially a operator= (NOT ==) for Sorted Arrays used by operator= in Container

inline void
equal (const Container<Type, Sorted_Array>& rhs)
Essentially a operator= (NOT ==) for Sorted Array Containers used by operator= in Container

inline Type*
intersection (Sorted_Array<Type> &a)
Find the first intersection element between this array and a

inline void
intersection (Sorted_Array<Type> &a,
Sorted_Array<Type> &b)
Fill array b with the entire intersection of elements between this & a.

inline Sorted_Array <Type> &
operator= (const Sorted_Array<Type>& sa)
Equals operator. For completness and to remove compilier warnings

1.8.1

Type& **operator()** (const Index)

Sorted arrays don't allow direct access to elements.

1.8.2

inline Index **insert** (const Type& item, const bool lock=true)

Return Value: s position that item was inserted to

1.8.3

inline Index **find_position** (const Type& item, bool * found) const

Finds the position where an item is or should be in the array

Finds the position where an item is or should be in the array. Doesn't lock the array - may be adversely affected if array resizes during execution.

1.8.4

~**Sorted_Array** ()

Default destructor

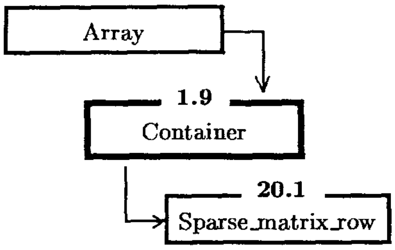
Default destructor. Deliberately not virtual. Does nothing, Heap_Array's destructor does the work.

1.9

template <class Type, class Array> class **Container** : public Array

Container is a decorator class that defines common operations for any container

Inheritance



Public Members

Container (const Index n = 0)

Default constructor. Construction passed to the Array class

Container (const Index n, const Index blksz, const Index jump, Type* init)

Construct array of n elements in blocks of given size

Container (const Container<Type, Array>& a)

Copy Constructor.

template <class B> inline bool
operator < (const B& rhs) const

< *comparison operator for Array container classes*

template <class B> inline bool
operator<= (const B& rhs) const

<= *comparison operator for Array container classes*

template <class B> bool
operator > (const B& rhs) const

> *comparison operator for Array container classes*

template <class B> bool
operator>= (const B& rhs) const

>= *comparison operator for Array container classes*

template <class B> bool

operator!= (const B& rhs) const
!= comparison operator for Array container classes

template <class Rhs_type> inline bool
operator == (const Rhs_type& rhs) const
<= comparison operator for Array container classes

inline Container <Type, Array> &
operator= (const Type val)
Assignment from a scalar

template <class Rhs_type> inline Container <Type, Array> &
operator= (const Rhs_type& rhs)
Container equals container operator=

inline Container <Type, Array> &
operator= (const Container<Type, Array>& rhs)
Our specialization required so some compilers don't override

inline **operator Sstring** ()
Allows containers to be represented as Sstrings

1.9.1 void **pack** (Send& buf) const
Support for any type of Container to be sent via Comm 34

void **un_pack** (Recv& buf)
Support for any type of Container to be recv via Comm.

~Container () *Destructor. Does nothing, Templatized arrays do the real destruction.*

Container is a decorator class that defines common operations for any container. Right now our only types of containers are the array classes. They can be used without being adorned by container, but this is not normally recommended as they can not then interact with other containers.

1.9.1

void **pack** (Send& buf) const

Support for any type of Container to be sent via Comm

Support for any type of Container to be sent via Comm. This won't pack in any "hidden" information such as references and block sizes, as they may be invalid on the other side and almost certainly can be rebuilt there. This provides the advantage that a container of one type can be received as a container of a different type, i.e. a stack array be received as an adjustable array.

2

(T,H) typedef long **Gid**

Integer large enough to provide a unique identifier for a large number of objects across a large number of processors

3 What's in Comm.H?

Names

In simple terms, Comm.H contains several classes that collectively act as a thread-safe MPI wrapper. By proper use of these classes, a user can send an arbitrary object from one processor to another. How can we communicate an arbitrary object when MPI only knows about int, double, char, etc.? We take a pass-the-buck approach: any object that wishes to be communicated must have member functions that enable the object to pack itself into a buffer or unpack itself out of a buffer. This packing and unpacking is recursive and eventually we recurse down to an intrinsic variable that MPI actually knowns how to deal with.

3.1 class **Communicator**

Communicator class to define more complicated topologies than just MPI_COMM_WORLD

Public Members

Communicator () *Default constructor generates a Communicator using MPI_COMM_WORLD*

Communicator (int n, int * procs)

3.1.1

Communicator (const Communicator& comm)

Copy constructor

Copy constructor. Creates a duplicate MPI communicator.

3.1.2

Communicator& **operator=** (const Communicator& comm)

Assignment operator

Assignment operator. Creates a duplicate MPI communicator.

3.1.3

int **nproc** ()

Number of processes in this communicator

Return Value: Number of processes

3.1.4

int **me** ()

My unique ID within this communicator

My unique ID within this communicator. Every MPI process is given a unique process ID per communicator, usually from `0 to (processes - 1). This ID is constant for the life of the communicator.`

Return Value: `Processor ID`

3.2

class **Comm**

Comm class

Public Members

dfile ()	*Every processor has its own file for writing diagnostic information* .	45

Comm class. This class is used to start an MPI process. Each process must instantiate one and only one Comm object. Instantiation of this object starts MPI. This class also has some miscellaneous static member functions for common communication operations.

3.2.1

Comm (int* p_argc, char** p_argv[])

Constructor

Constructor. This is the only constructor. Instantiation starts MPI. Note that MPI needs both argc and argv, and that these may be modified. So if you don't want them modified, make a copy first.

Parameters: p_argc — Pointer to argc
p_argv — Pointer to argv

3.2.2

~Comm ()

Destructor

Destructor. The destructor stops MPI.

3.2.3

void **nop** ()

Don't do a darn thing

Don't do a darn thing. Use this to eliminate "never referenced" warnings.

3.2.4

static int **nproc** ()

Number of processes

Number of processes. A SPMD paradigm is assumed, where there are N processes running the same executable. The number of processes is constant for the life of the program.

Return Value: Number of processes

3.2.5

static int **me** ()

My unique ID

My unique ID. Every MPI process is given a unique process ID, usually from 0 to (processes - 1). This ID is constant for the life of the program.

Return Value: Processor ID

3.2.6

static const char* **name** ()

Every MPI process is given a unique name (character string)

Every MPI process is given a unique name (character string). The name depends upon the MPI implementation.

Return Value: A constant character string.

3.2.7

static int **barrier** (Communicator& comm=default_comm)

Synchronize all processes

Synchronize all processes. When a process invokes barrier, it waits until all other processes invoke barrier.

3.2.8

template <class Valtype> static Valtype **max** (Valtype loc_val)

Find the global maximum for some value

Find the global maximum for some value. The value is templated and the templated type must support the comparison operators. This must be called exactly once for every process in the array. Alternatively, try the global_max(T data, T* reduced_data) function. It's probably faster, but I'm not sure it works for non-intrinsic types.

3.2.9

template <class Valtype> static Valtype **min** (Valtype loc_val)

Find the global minimum for some value

Find the global minimum for some value. The value is templated and the templated type must support the comparison operators. This must be called exactly once for every process in the array. Alternatively, try the global_min(T data, T* reduced_data) function. It's probably faster, but I'm not sure it works for non-intrinsic types.

3.2.10

template <class Valtype> static Valtype **sum** (Valtype loc_val)

Find the global sum for some value

Find the global sum for some value. The value is templated and the templated type must support the arithmatic operators. This must be called exactly once for every process in the array. Alternatively, try the global_sum(T data, T* reduced_data) function. It's probably faster, but I'm not sure it works for non-intrinsic types.

3.2.11

static Gid **global_offset** (Gid my_size, Communicator& comm=default_comm)

Find my global offset with respect to all the processors with a rank below mine

Find my global offset with respect to all the processors with a rank below mine. That is, if P0 has a size of 3, P1 has a size of 5 and P2 has a size of 4, P0 gets a global offset of 0 (because there's nothing before it in rank), P1 gets a global offset of 3 (size of P0) and P2 gets a global offset of 8 (size of P0 + P1).

3.2.12

static Gid* **global_cutoffs** (Gid my_size, Communicator& comm=default_comm)

Get the list of cutoffs between all the processors

Get the list of cutoffs between all the processors. That is, if P0 has a size of 3, P1 has a size of 5 and P2 has a size of 4, the global division array will be [2, 7, 11] which means that P0 owns 0-2, P1 owns 3-7 and P2 owns 8-11, so the number stored for each processor is the upper bound of the elements that it owns. Oh, and delete the array when you're done with it.

3.2.13

static double **time** ()

Everybody always wants timing information

Everybody always wants timing information. There are several different timing mechanisms, unfortunately none seems to work consistently across all platforms. Right now this method simply invokes the standard C clock utility instead of the MPI timing routine. The MPI timing routine seemed to give wall clock time, which is not very useful when sharing a processor with a dozen other users. This time method returns CPU time. The clock resolution and maximum timing interval are system dependent.

Return Value: The current time, in seconds.

3.2.14

static ofstream& **dfile** ()

Every processor has its own file for writing diagnostic information

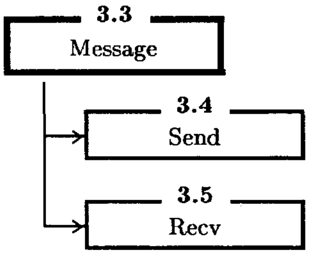
Every processor has its own file for writing diagnostic information. The file is created automatically in Comm's constructor. This is a little cleaner (and faster) than having everybody write diagnostic info to cout.

Return Value: An open ofstream object

3.3 class **Message**

Base class Message

Inheritance



Public Members

		Message (Communicator& comm=default_comm)	*Default Constructor*	
		Message (const Message &)	*Copy Constructor*	
	Message&	**operator=** (const Message &)	*Assignment operator*	
	virtual	~**Message** ()	*Basic Destructor*	
3.3.6	int	**length** () const	....	46
3.3.7	int	**length** (int at_least_this_big)	*Allows the user to manually increase the size of the buffer*	47
3.3.8	int	**amount** () const	....	47

Protected Members

Base class Message. This is a base class that contains data and methods common to both Send and Recv. Do not attempt to instantiate a Message directly, compiler will not let you do this.

3.3.6

int **length** () const

Return Value: The current length of the buffer.

3.3.7

int length (int at_least_this_big)

Allows the user to manually increase the size of the buffer

Return Value:	The new length of the buffer.
Parameters:	`at_least_this_big` — The buffer size will be >= this param.

3.3.8

int amount () const

Return Value:	The amount of data (bytes) in the buffer.

3.3.1

Communicator& communicator

The communicator we're using for this message

The communicator we're using for this message. If the user doesn't set one, it defaults to default_comm (MPI_COMM_WORLD).

3.3.2

char* **buf**

The buffer

The buffer. All data is converted to bytes and put in the buffer. The buffer grows automatically as needed.

3.3.3

int **my_tag**

The the message tag

The the message tag. Usually an integer 0 to M, see MPI documentation for more details.

3.3.4

MPI_Status* **status**

The status of the message

The status of the message. Some MPI calls have a status data structure (such as MPI_Iprobe), but users should not need to access this data directly.

3.3.5

MPI_Request* **request**

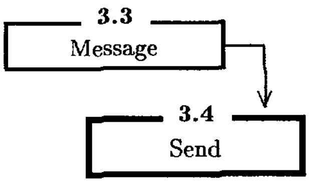
Some MPI calls have a request data structure, such as non-blocking send and receive calls

Some MPI calls have a request data structure, such as non-blocking send and receive calls. Users should not need to access this data directly.

3.4 class **Send** : public Message

Send buffer

Inheritance



Public Members

		Send (Communicator& comm = default_comm)	*Basic Constructor*	
		~Send ()	*Basic Destructor*	
3.4.2	Send&	**operator()** (const int& the_destination, const int& the_tag = DEFAULT_TAG)	*Buffer set-up*	51
3.4.3	Send&	**send** ()	*Send the buffer*	51
3.4.4	Send&	**send** (const int& the_tag)	*Same as send() except that the_tag is used as the tag*	52
3.4.5	Send&	**ready_send** ()	*Send the buffer*	52
3.4.6	Send&	**ready_send** (const int& the_tag)	*Same as ready_send() except that the_tag is used as the tag*	53
3.4.7	Send&	**sync_send** ()	*Send the buffer using an MPI synchronous send*	53
3.4.8	Send&	**sync_send** (const int& the_tag)	*Same as sync_send() except that the_tag is used as the tag*	53
3.4.9	Send&	**post_send** ()	*Post a send*	54
3.4.10	Send&	**post_send** (const int& the_tag)		

Send buffer. A Send object can be considered a smart buffer in the sense that this buffer knows how to send itself to another process. A program may have as many Send buffers as it wants, but keep in mind that these buffers do take up memory. It is not necessary to create a new buffer for every message, the Send buffer can be used over and over again. The idea is that the program packs several arbitrary objects (not necessarily of the same type) into the Send buffer, and the tells the buffer to send itself. Kind of like putting several christmas gifts into one big box, and then sending the box to a single destination.

This class uses the MPI_Pack routine to pack data into the buffer, and all messages are sent using MPI_PACKED data type. This is not the fastest way to do message passing, but it is the most general. This is very useful for dynamic objects, i.e. objects that shrink and grow during the life of the program.

3.4.2

Send& **operator()** (const int& the_destination, const int& the_tag = DEFAULT_TAG)

Buffer set-up

Buffer set-up. This operator sets the other to the_destination, sets my_tag to the_tag, and sets ok_to_pack to TRUE. The destination is a processor ID in the range 0 .. (num processes-1). The tag is an integer in the range 0 .. M (M is implementation specific, but it is usually the maximum unsigned int). The user must invoke this method this prior to packing data into the buffer. In order to do a broadcast (send to everybody except myself) the destination may be BROADCAST. Note that the user does not have to specify the tag here, it could be specified using the send method.

Return Value:	Reference to the Send buffer.
Parameters:	`the_destination` — The destination process. `the_tag` — Tag used to tag the outgoing message.

3.4.3

Send& **send** ()

Send the buffer

Send the buffer. The buffer is sent to other using my_tag. A MPI basic send is used, this is the best all around send. Upon return the buffer is free for reuse, but the message might not have been received yet, i,e. it could have been buffered by the system. Note that some systems have small buffers (kilobytes) and if the message exceeeds the buffer, this call will block until the destination does a receive. Look at the test suite for demonstration of how to use the send/recv methods in a deadlock-free manner.

Return Value: Reference to the Send buffer.

3.4.4

Send& **send** (const int& the_tag)

Same as send() except that the_tag is used as the tag

Return Value: Reference to the Send buffer.

3.4.5

Send& **ready_send** ()

Send the buffer

Send the buffer. This method uses an MPI ready send. The buffer is free for re-use upon return. Do not use this method unless you are absolutely, positively sure that the destination process has already posted a receive. If you are sure that the receive has been posted, this method can be very fast. It does not use any system buffers. This method requires a significant amount of process synchronization and user sophistication, use at your own risk!

Return Value: Reference to the Send buffer.

3.4.6

Send& **ready_send** (const int& the_tag)

Same as ready_send() except that the_tag is used as the tag

Return Value: Reference to the Send buffer.

3.4.7

Send& **sync_send** ()

Send the buffer using an MPI synchronuous send

Send the buffer using an MPI synchronuous send. This method does not return until the corresponing receive has started (but not necessarily finished). Otherwise it is the same as the basic send().

Return Value: Reference to the Send buffer.

3.4.8

Send& **sync_send** (const int& the_tag)

Same as sync_send() except that the_tag is used as the tag

Return Value: Reference to the Send buffer.

3.4.9

Send& **post_send** ()

Post a send

Post a send. This method tells the system "Send this buffer when you get a chance, and let me do some real work while you are sending the buffer." This method returns (almost) immediately, allowing the calling program to do something else while the message is being sent. However the buffer is locked, the user cannot re-use the buffer. This is useful for overlapping real work with communication, which is required in order to achieve optimal performance. But it only makes a difference for large messages. At some point the user must invoke complete_send() to verify that the buffer has been send and unlock the buffer for re-use.

Return Value: Reference to the Send buffer.

3.4.10

Send& **post_send** (const int& the_tag)

Same as post_send() except that the_tag is used as the tag

Return Value: Reference to the Send buffer.

3.4.11

Send& **post_ready_send** ()

This is a combination of post_send() and ready_send()

This is a combination of post_send() and ready_send(). You can consider yourself an parallel guru if you sucessfully use this method in a real application.

Return Value: Reference to the Send buffer.

3.4.12

Send& **post_ready_send** (const int& the_tag)

Same as post_ready_send() except that the_tag is used as the tag

Return Value: Reference to the Send buffer.

3.4.13

Send& **post_sync_send** ()

This is a combination of post_send() and sync_send()

This is a combination of post_send() and sync_send(). This method returns (almost) immediately, allowing the calling program to do something else while the message is being sent. However the buffer is locked, the user cannot re-use the buffer.

Return Value: Reference to the Send buffer.

3.4.14

Send& **post_sync_send** (const int& the_tag)

Same as post_sync_send() except that the_tag is used as the tag

Return Value: Reference to the Send buffer.

3.4.15

Send& complete_send ()

This method completes the send operation initiated by a post send

This method completes the send operation initiated by a post send. Upon return the buffer is unlocked and is free for re-use. The message may or may not have arrived at the destination, it depends upon which version of post send was invoked.

Return Value: Reference to the Send buffer.

3.4.16

Send& broadcast ()

Send this buffer to everybody execpt me

Send this buffer to everybody execpt me. Note that this method does not use an MPI broadcast! This method basically does a send to everybody. The destination processes simply does an everyday receive. This is slower than the real MPI broadcast but it is more general, we can tag the message, we can probe it, etc. In order to do a brodcast the user must set the destination to BROADCAST. Note that this method might not work properly in a hetero environment. The typical use of this method is when process 0 reads in some data from a file and broadcasts it to everybody else. In this case speed is less important than generality.

Return Value: Reference to the Send buffer.

3.4.17

Send& broadcast (const int& the_tag)

Same as broadcast() except that the_tag is used as the tag

Return Value: Reference to the Send buffer.

3.4.18

int **dest** () const

Return Value: Retruns the dest process ID

3.4.19

int **tag** () const

Return Value: Returns the message tag

3.4.20

template <class T> void **pack** (T& object)

Templated pack function

Templated pack function. This method packs the object of type T into the buffer. Every non-intrinsic object must know to pack itself into the buffer. If the object is an instrinsic (char, double, etc.) than it is simply packed into the buffer using the MPI pack function. If the object is not an intrinsic, than the object must have its own pack method of the form void pack(&Send buf). The objects pack method can in turn use this pack method to pack its data. This way we can create and modify classes ad infinitum without ever modifying this communication class.

Parameters: `object` — The object to be packed into the buffer.

3.4.21

template <class T> void **pack** (T* object, int n)

Same as pack(), except we pack n objects into the buffer

Parameters: `object` — Pointer to an array of objects
`n` — The number of objects

3.4.22

template <class T> Send& **operator<<** (const T& data)

This operator simply invokes the pack() method

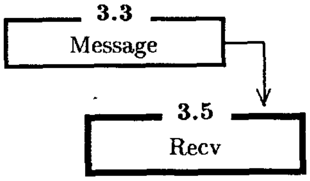
This operator simply invokes the pack() method. The idea is to make communication look like I/O. See the example program.

Parameters: `p` — A referecne to this object
`data` — The data object to pack into the buffer

3.5 class **Recv** : public Message

Receive buffer

Inheritance



Public Members

Recv (Communicator& comm=default_comm)
Basic Constructor

~**Recv** () *Basic Destructor*

3.5.1 Recv& **operator)** (const int& the_source = ANY_SOURCE, const int& the_tag = ANY_TAG)
This receive operation blocks until it receives a message from the_source with the_tag 60

3.5.2 Recv& **post_recv** (const int& the_source = ANY_SOURCE, const int& the_tag = ANY_TAG)
This is similar to operator() 61

3.5.3 Recv& **complete_recv** () *This method completes the receive operation posted by post_recv()* .. 61

3.5.4 int **source** () const 62

3.5.5 int **tag** () const 62

3.5.6 int **size** () const 62

3.5.7 template <class T> void
un_pack (T& object)
Templated unpack function 63

3.5.8 template <class T> void

un_pack (T* object, int n)
Similar to un_pack, except this method unpacks an array of objects 63

3.5.9 template <class T> Recv&
operator>> (T& data)
This method simply calls un_pack() 64

void **pack** (Send& buf) const
Allow the buffer to pack itself into a Send buffer without ever unpacking

Receive buffer. A Recv object can be considered a smart buffer in the sense that this buffer knows how to receive itself from another process. A program may have as many Recv buffers as it wants, but keep in mind that these buffers do take up memory. It is not necessary to create a new buffer for every message, the Recv buffer can be used over and over again. The idea is that the Recv buffer is like a big box full of little presents. You invoke a receive method to receive the box, and then you unpack the presents one by one. The presents are arbitrary objects, not necessarily of the same type.

This class uses the MPI_Unpack routine to unpack data from the buffer, and all messages are received using MPI_PACKED data type. This is not the fastest way to do message passing, but it is the most general. This is very useful for dynamic objects, i.e. objects that shrink and grow during the life of the program.

3.5.1

Recv& **operator)** (const int& the_source = ANY_SOURCE, const int& the_tag = ANY_TAG)

This receive operation blocks until it receives a message from the_source with the_tag

This receive operation blocks until it receives a message from the_source with the_tag. After the message arrives you can invoke source() to see who it came from, tag() to examine the tag, and size() to see how big it is.

Return Value: Reference to this object.

3.5.2

Recv& **post_recv** (const int& the_source = ANY_SOURCE, const int& the_tag = ANY_TAG)

This is similar to operator()

This is similar to operator(). This method posts a recieve. This method tells the system "Receive this buffer when you get a chance, and let me do some real work while you are waiting for the buffer." This method returns (almost) immediately, allowing the calling program to do something else while the message is being received. However the buffer is locked, the user cannot attempt to unpack anything from the buffer yet. This method is useful for overlapping real work with communication, which is required in order to achieve optimal performance. But it only makes a difference for large messages. At some point the user must invoke complete_srecv() to verify that the buffer has been received and set ok_to_unpack to TRUE.

Return Value: Reference to this object.

3.5.3

Recv& **complete_recv** ()

This method completes the receive operation posted by post_recv()

This method completes the receive operation posted by post_recv(). It is blocking, the method will not return return until a message with the correct tag from the correct source arrives. This method sets ok_to_unpack to TRUE, user can unpack data from the Recv buffer upon return.

Return Value: Reference to this object.

3.5.4

int **source** () const

Return Value: Retruns the source process ID

3.5.5

int **tag** () const

Return Value: Returns the message tag

3.5.6

int **size** () const

Return Value: Returns the size of the message, in bytes

3.5.7

template <class T> void **un_pack** (T& object)

Templated unpack function

Templated unpack function. This method unpacks the object of type T from the buffer. Every non-intrinsic object must know to unpack itself from the buffer. If the object is an instrinsic (char, double, etc.) than it is simply unpacked from the buffer using the MPI unpack function. If the object is not an intrinsic, then the object must have its own unpack method of the form void unpack(&Recv buf). The objects unpack method can in turn use this unpack method to unpack its data. This way we can create and modify classes ad infinitum without ever modifying this communication class.

Parameters: `object` — The object to unpack

3.5.8

template <class T> void **un_pack** (T* object, int n)

Similar to un_pack, except this method unpacks an array of objects

Parameters: `object` — A pointer to an array of objects
`n` — The number of objects to unpack

3.5.9

template <class T> Recv& **operator>>** (T& data)

This method simply calls un_pack()

This method simply calls un_pack(). It is used so that communication looks just like I/O. See the example program.

Parameters: `p` — A reference to this object
`data` — The data object to unpack

4

template <class Type, class ArgType> void* **cloner** (void * our_void_arg)

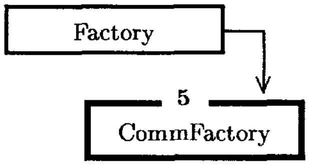
Function to run in thread to handle cloning on all procs

5

template <class Type, class ArgType> class **CommFactory** : public Factory<Type, ArgType>

CommFactory is an extention to Factory that ensures that the clone of the given object is cloned across all processors such that it exists uniquely on each processor

Inheritance



Public Members

	void initialize			
		(**const int ctb, Mutex * lock**) (const int ctb, Mutex *)	*Call this before using this CommFactory for cloning*	
	void	**finalize** ()	*Call this before exiting*	
5.1	Type*	**clone** (ArgType& arg)	*Get a pointer to the unique instance of the given type defined by the given argument type*	67
	void	**clone_local** (ArgType& arg, const Index pos)	*Does the actual construction on each processor*	

5.1

Type* **clone** (ArgType& arg)

Get a pointer to the unique instance of the given type defined by the given argument type

Get a pointer to the unique instance of the given type defined by the given argument type. If it doesn't exist, it will be created on all processors.

Return Value:	s Pointer to the unique instance
Parameters:	The — argument that uniquely defines the requested instance

6

template <class Type, class ArgType> void* **cloner** (void * our_void_arg)

This is the function used in the thread that CommFactory::initialize() spins off

This is the function used in the thread that CommFactory::initialize() spins off. On processor 0, it receives requests from arbitrary processors in serial fashion, returns if it already exists and tells the thread on all other processors to actually build the thing if it doesn't.

7

template <class Container, class Type> class **Free_list**

Free_list is a small helper class that can be used with Container classes that have items large enough to store at least a pointer to another item

Public Members

7.2		**Free_list** (Container& ct, bool gc)	*Constructs a Free_list initializing it with a reference to container* .	71
		Free_list (const Free_list& rhs)	*Default copy constructor. Should not be called.*	
7.3	void	**turn_on_auto_garbage_collection** ()	*Start the auto garbage collecter and immediately perform any required garbage collection before returning*	72
	inline void	**turn_off_auto_garbage_collection** ()	*Turn off the automatic garbage collecter and use the free list mechanism*	
	inline void	**turn_on_free_list_usage** ()	*Turns on free list useage when obtaining next() item(s)*	
7.4	inline void	**turn_off_free_list_usage** ()	*Turns off free list useage when obtaining next() item(s)*	72
	inline bool	**free_list_usedQ** ()	*Returns a boolean flag of whether or not the free_list is being used for next() operations*	
	Free_list&	**operator=** (const Free_list& rhs)	*Equals operator.*	
	bool	**assert_test** ()	*Test that size and alignment meet criteria for using a free list*	

7.5	inline void	**add_to_free** (Type* item)	*Add item to free_list*	72
	inline Type*	**next** ()	*Returns item on free_list or next available item space in container*	
7.6	size_t	**next_block** (size_t num)	*Obtains a block of items with contiguous indicies from the container and returns the index to the first item*	72
	void	**reinitialize** ()	*Re_initializes the container and free list*	
	inline size_t	**freelist_size** ()	*Returns the number of items on the free_list*	
	inline size_t	**num_items** ()	*Returns the number of items in the container not counting those on the free_list*	

Free_list is a small helper class that can be used with Container classes that have items large enough to store at least a pointer to another item.

At present, ref(i), last(), block_size(), jump(), size(), index(Type*), and relocate(Type*) member functions are required by the templated container class. The ref(i) returns a reference to the i'th item in the array. The last() member function that returns a reference to the next available item in the container. The block_size() member function returns the number of items that fit in a given block of memory. The jump() member function specifys the jump byte size between successive items contained with a block. The size() member function provides the number of items currently in the container. The index(Type*) member function returns the index position given an item address. Lastly, relocate(Type*) member function is called by the Free list if it relocates an item.

There is a boolean garbage_collection flag that can be turned on to automatically perform garbage_collection on the container. Any items on the free list are immediately garbage collected.
If the notification of a data item is not important then the relocate(Type*) member function can be a no op. Garbage collection will move items from the end of the array to fill any vacancies

earlier in the list.

Caveat:

This is not meant for Containers that can move an item's location as the array size changes. At present this class should be used in conjunction with the Reference_Nc_Array. The garbage collection process assumes memory is laid out in blocks like the Reference_Nc_Array. If memory is contiguous then the array capacity could never change size. If a user wants to use a free_list to manage a contiguous memory situation then one way to accomplish this is to have a huge block_size for the Reference_Nc_Array. Also, useage of this Free list assumes the user is only adding through last() or the Free_list next() and add_to_free() member functions.

This Free_list class was built as a convenience mechanism to be used in association with Reference_Nc_arrays. It is not a general purpose garbage collector. The user can easily trash the Free_list if they only use it partially and sometimes use the Container to change size etc... You have been warned. A general purpose free_list, garbage_collection utility is far beyond the perview of this class.

7.2

Free_list (Container& ct, bool gc)

Constructs a Free_list initializing it with a reference to container

Constructs a Free_list initializing it with a reference to container. The auto_garb_collection boolean is set by the bool gc.

7.3

void **turn_on_auto_garbage_collection** ()

Start the auto garbage collecter and immediately perform any required garbage collection before returning

Start the auto garbage collecter and immediately perform any required garbage collection before returning. The use_free_list boolean is also set to true.

7.4

inline void **turn_off_free_list_usage** ()

Turns off free list useage when obtaining next() item(s)

Turns off free list useage when obtaining next() item(s). New item space is acquired from the end of the Container. Automatic garbage collection flag is set to false.

7.5

inline void **add_to_free** (Type* item)

Add item to free_list

Add item to free_list. If automatic garbage collection boolean is set then garbage collection is performed immediately.

7.6

size_t **next_block** (size_t num)

Obtains a block of items with contiguous indicies from the container and returns the index to the first item

Obtains a block of items with contiguous indicies from the container and returns the index to the first item. Garbage collection is done prior to obtaining the block from the container.

8

What's in Memory_pool.H?

Names

Memory_pool is a power of 2 queue memory allocator that provides memory always aligned to the requested size 8.4

Our aligned memory manager. The Memory_pool class gets raw aligned blocks of memory from the global new operator. This Memory_pool class is meant to be inherited by other Memory manager classes of higher functionality. The Memory_stamp class is more of a helper class for memory Pool and could be a nested class.

We're not interested in reinventing the wheel here. This is not meant to be a general all encompassing memory pool allocater.

This is a special purpose memory manager that always returns memory aligned on the size requested. All requests are increased to: nearest power of 2 >= (size requested + sizeof(size_t)).

The intended usage of this class is for large memory allocations based on integer powers of 2.

Specifically, the Memory_pool is not currently designed to handle:

1) Very small blocks of memory (few words or less)

2) Memory sizes far from an integer power of 2. For example, 1.5MB size.

3) Large numbers of small to medium sized allocated memory requests which really don't require any memory alignment restrictions

At a minimum each block requires:

sizeof(sizeof(size_t) + 2sizeof(void))

Actual memory usage overhead:

while memory is in use: sizeof(size_t):

while memory is on free list: sizeof(sizeof(size_t) + 2sizeof(void)):

to manage huge system block calls: Small amount if incidental pointers etc...

Caveat:

At present we don't have the time to be extremely elegant. To ease code writing, we will burden the user of these classes with the following:

sizeof(size_t) will be added to every request for memory before calculating power of 2 size! This means that it will be optimal to ask for say 1024-sizeof(size_t) bytes rather than 1024 bytes. Requesting 1024 bytes will require 2048 bytes of memory.

Concerns:

Memory management deals with many memory alignment portability issues.

Comments:

At a slight performance penalty the prev pointer used when memory is on the free_list could be eliminated by rewriting put_in_use() and put_on_free() member functions to not prev. This would reduce the required minimum size to be 2sizeof(void). So even using 64 bit pointers the minimum real size would be 16 bytes. This would be comparable to normal system restrictions put_in_use() would need handle to loop thru free_list to find prev address.

Another approach whould be to use the size_t mask when on free_list but this has many software complexity issues such as knowing the "best_alignment" and "last" bit flags.

8.1

struct **Prev_next**

Prev_next struct is a Memory_stamp number with prev, next pointers to create a doublely linked list

Members

size_t	**num**	*The Memory stamp*

Stamp	**next_**	*Pointer to next memory location*
Stamp	**prev_**	*Pointer to previous memory location*
	Prev_next ()	*Default constructor.*
	Prev_next (const Prev_next&)	*Copy Constructor. For completness and to remove compilier warnings*
Prev_next&	**operator=** (const Prev_next&)	*Equals operator. For completness and to remove compilier warnings*

Prev_next struct is a Memory_stamp number with prev, next pointers to create a doublely linked list. This structure represents the memory usage while memery is on the free list.

8.2 class **Memory_stamp**

The Memory_stamp class manages the writing and reading of the memory stamp for the Memory pool

Public Members

	Memory_stamp (Stamp v = 0)	*Default Constructor.*
	Memory_stamp (const Memory_stamp& rhs)	*Copy Constructor. For completness and to remove compilier warnings.*
size_t	**p2_size** ()	*Return the power of 2 for this memory size*
bool	**in_use** ()	*Return boolean flag of whether memory is in use*

	bool	**best_align** ()	*Return boolean flag of whether this memory is already aligned on best possible boundary*	
	bool	**last** ()	*Return boolean flag of whether this is the last memory chunk of a huge block*	
	void	**set_in_use** ()	*Set the "in use?" boolean flag*	
	void	**unset_in_use** ()	*Unset the "in use?" boolean flag*	
	void	**set_best_align** ()	*Set the "best alignment?" boolean flag*	
	void	**unset_best_align** ()	*Unset the "best alignment?" boolean flag*	
	void	**set_last** ()	*Set the "last?" boolean flag*	
	void	**unset_last** ()	*Unset the "last?" boolean flag*	
	void	**set_size** (size_t n)	*Set the size in powers of 2 for this memory chunk*	
	void	**set_stamp** (size_t n)	*Set the stamp to a given number*	
8.2.2	size_t	**make_number** (Stamp v)	*Create a number that can be used for alignment calculations given a Stamp*	78
	size_t	**stamp_num** ()	*Return the Stamp number*	
	Stamp	**prev** (Stamp v)	*Return the previous memory location*	
	Stamp	**next** (Stamp v)	*Return the next memory location*	
	void*	**use_address** ()	*Return the address given to user for this stamp*	
	Stamp	**address** ()	*Return address of stamp*	
8.2.3	Stamp	**next_address** ()	*Read Stamp size and return the Stamp of memory below us The caller is responsible for determining whether the returned Stamp is valid*	78

8.2.4	Stamp	**prev_address** ()	*Read Stamp size and return the Stamp of memory above of us The caller is responsible for determining whether the returned Stamp is valid*	79
	void	**set** (Stamp v)	*Bind the stamp to new location*	
	inline Stamp	**put_in_use** (Stamp v)	*Take this block of memory off the free list This is only called with memory currently on free_list next_ is set to prev_ and given address's prev_->next ptr is set to given address's next ptr*	
8.2.5	inline void	**put_on_free** (Prev_next& free_list)	*Put on the free list*	79
	Memory_stamp&	**operator=** (const Memory_stamp& rhs)	*Equals operator. For completness and to remove compilier warnings*	

Protected Members

b	static const size_t	**size_mask**	*Enough size bits for 2;255*	??
ytes of memory	static const size_t	**in_use_**	*Bit toggle flag specifing whether memory is being used*	
8.2.1	static const size_t	**best_align_**	*Bit toggle flag specifing whether memory alignment is best possible*	79
	static const size_t	**last_**	*Bit toggle flag specifing when this memory is the last chunk of memory in a huge block retrieved by the system*	
	size_t	**num**	*The Memory stamp number*	
	Stamp	**next_**	*Pointer to next memory location*	

Stamp	**prev_**	*Pointer to previous memory location*
Stamp	**ptr**	*Pointer to actual Stamp ed*

The Memory_stamp class manages the writing and reading of the memory stamp for the Memory pool. The memory stamp is composed of a size_t* word placed in just above the memory address given back to the user and 2 void** pointers that point to prev and next memory when on free list. The Memory_stamp has all knowledge about size, alignment, and position relative to other memory etc... but has no knowledge of actual type or intended usage.

8.2.2

size_t **make_number** (Stamp v)

Create a number that can be used for alignment calculations given a Stamp

Create a number that can be used for alignment calculations given a Stamp. The alignment will be based on memory address after the Stamp number since we base alignment of user's handle to memory and not actual memory alignment

8.2.3

Stamp **next_address** ()

Read Stamp size and return the Stamp of memory below us The caller is responsible for determining whether the returned Stamp is valid

Read Stamp size and return the Stamp of memory below us The caller is responsible for determining whether the returned Stamp is valid. Memory_stamp only has knowledge local size jumps.

8.2.4

Stamp **prev_address** ()

Read Stamp size and return the Stamp of memory above of us The caller is responsible for determining whether the returned Stamp is valid

Read Stamp size and return the Stamp of memory above of us The caller is responsible for determining whether the returned Stamp is valid. Memory_stamp only has knowledge local size jumps.

8.2.5

inline void **put_on_free** (Prev_next& free_list)

Put on the free list

Put on the free list. Only ptr and num are setup prev_, next_ are not valid by design

8.2.1

static const size_t **best_align_**

Bit toggle flag specifing whether memory alignment is best possible

Bit toggle flag specifing whether memory alignment is best possible. If this bit is not set then memory can potentially be combined into a bigger contiguous block

8.3

struct **Sys_block**

Sys_block struct is for use as a simple linked list inside the Memory_pool class

Members

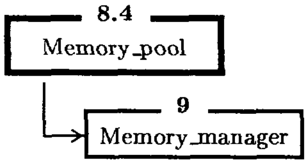
Sys_block*	**next**	*Pointer to next block in linked list*
void*	**address**	*Actual address to top of huge system retrieved block*
	Sys_block ()	*Default Constructor.*
	Sys_block (const Sys_block& sb)	*Copy Constructor. For completness and to remove compilier warnings*
Sys_block&	**operator=** (const Sys_block& rhs)	*Equals operator. For completness and to remove compilier warnings*

Sys_block struct is for use as a simple linked list inside the Memory_pool class. The linked list represents all blocks retrieved by the system that are currently in use or on free list. This class could be nested class inside Memory_pool but alas some compiliers are behind the times.

8.4 class **Memory_pool**

Memory_pool is a power of 2 queue memory allocator that provides memory always aligned to the requested size

Inheritance



Public Members

	static Memory_pool*	**clone** ()	*Creates a unique instance of the Memory_pool class using a Singleton pattern*	
8.4.7	bool	**legal_address** (void *user_address)	*Check if given address is a legal address that user is currently allowed to release back too use*	82
	inline void*	**memory** (size_t siz)	*Provide memory to user of at least siz+sizeof(size_t) bytes alignment guaranted to be >= the total size*	
	void	**release** (void *v)	*Return the memory back to free list combining the memory into larger chunks if possible to prevent memory fragmentation*	
	virtual	**~Memory_pool** ()	*Default destructor gives memory back to system regardless of whether the memory has been released*	
	void	**report** (ostream& out)	*Report out diagnostics about present memory pool usage*	

Protected Members

	Memory_pool ()	*Default constructor*

Memory_pool is a power of 2 queue memory allocator that provides memory always aligned to the requested size. Memory_pool retrieves large blocks of memory from system and splits this memory into aligned chunks of memory. As requests for memory are processed larger chunks of memory are split even further and provided to the user. As memory is released back to the pool it is combined back into larger chunks.

This is a special purpose memory manager that always returns memory aligned on the size requested. All requests are increased to:

```
   nearest power of 2 >= (size requested + sizeof(size_t)).


 The intended usage of this class is for large memory
 allocations based on integer powers of 2.



 Specifically, the Memory_pool is not currently designed
 to handle:

   1. Very small blocks of memory (few words or less)
   2. Memory sizes far from an integer power of 2.
      For example, 1.5MB size.
   3. Large numbers of small to medium sized allocated
      memory requests which really don't require any
      memory alignment restrictions
```

8.4.7

bool **legal_address** (void *user_address)

Check if given address is a legal address that user is currently allowed to release back too use

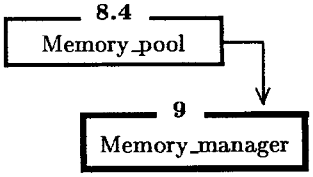
Check if given address is a legal address that user is currently allowed to release back too use. The intended usage is a a checking device for memory. Only call this function if it is ok to send a message to cerr stating the user_address is not legal.

9

class **Memory_manager** : public Memory_pool

Memory management class

Inheritance



Public Members

Memory_manager (const Memory_manager& self)
Calling The copy constructor is explicitly disabled since we are a Singleton class

static Memory_manager*
clone () *Creates a unique instance of the Memory_manager class using a Singleton pattern*

static size_t
delete_clone () *Destruct the clone*

9.4 inline void
update_newed (const size_t amount)
Update the amount of memory on the heap 84

9.5 inline void
update_deleted (const size_t amount)
Update the amount of memory on the heap 85

9.6 inline void
update_popped (const size_t amount)
Update the amount of memory on the stack 85

9.7 inline void

update_pushed (const size_t amount)
Update the amount of memory on the stack 85

inline void*
new_aligned (size_t size)
Obtain memory from Memory_pool aligned on a power of 2 associated with the requested size

inline void
release_aligned (void *v, const size_t size)
Released memory obtained by calling new_aligned member function

void **report** (ostream& os)
Print out the memory usage statistics to the given output ostream

9.8 virtual **~Memory_manager** ()
Default destructor 85

Memory management class. This is a global singelton class. It keeps track of how much memory we have used. It keeps track of both stack memory and heap memory. File Memory_manger.H

9.4

inline void **update_newed** (const size_t amount)

Update the amount of memory on the heap

Parameters: **amount** — The amount of memory just added by new.

9.5

inline void **update_deleted** (const size_t amount)

Update the amount of memory on the heap

Parameters: amount — The amount of memory just deleted by delete

9.6

inline void **update_popped** (const size_t amount)

Update the amount of memory on the stack

Update the amount of memory on the stack. The class or method that allocates stack memory needs to call this.

Parameters: amount — The amount of memory just added.

9.7

inline void **update_pushed** (const size_t amount)

Update the amount of memory on the stack

Update the amount of memory on the stack. The class or method that allocates stack memory needs to call this.

Parameters: amount — The amount of memory just deleted.

9.8

virtual ˜**Memory_manager** ()

Default destructor

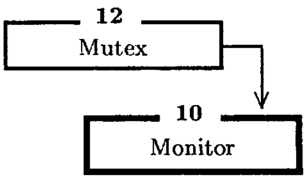
Default destructor. Only deletes the Memory_manager for the last reference.

10

class **Monitor** : public Mutex

This allows us to track useage of a critical section and guarentee exclusive access when needed

Inheritance



Public Members

inline	**Monitor** ()	*Default constructor: initializes the Monitor*
inline	**~Monitor** ()	*Default destructor: destroys the Monitor*
inline void	**lock** ()	*Request exclusive access to this Monitor*
inline Monitor&	**operator++** ()	*Request non-exclusive access to this Monitor*
inline Monitor&	**operator++** (int)	*Request non-exclusive access to this Monitor*
inline Monitor&	**operator–** ()	*Inditate that we are finished with non-exclusive access to this Monitor*
inline Monitor&	**operator–** (int)	*Inditate that we are finished with non-exclusive access to this Monitor*

This allows us to track useage of a critical section and guarentee exclusive access when needed. This is called a semaphore in some circles, but (unfortunately), POSIX semantics consider a semaphore to be something quite different. This is particularly useful if you have many threads reading some data, but you want to guarentee that you're the only thread writing the data. This is accomplished as follows:

```
Reading Thread   |   Writing Thread
-----------------+--------------------
monitor++;       |   monitor.lock();
data.read();     |   data.write();
monitor--;       |   monitor.unlock();
```

see, there can be numerous threads reading at once, but as soon as a writing thread comes along, no other readers can enter the critical section until the writer is done. Likewise, the writer can't start until all the readers have left.

11

#define **Static**

If we have threads then the word "Static" will be " " else it will be "static"

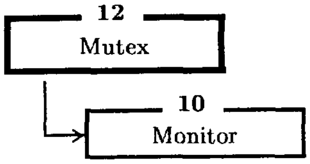
If we have threads then the word "Static" will be " " else it will be "static". This will allow us to turn on and off the use of temporaries based on threads

12

class **Mutex**

Provide a basic thread locking mechanism

Inheritance



Public Members

12.1	**Mutex** ()	*It is assumed we are running the POSIX_THREADS*	89
	Mutex (const Mutex&)	*Copy constructor*	

12.1

Mutex ()

It is assumed we are running the POSIX_THREADS

It is assumed we are running the POSIX_THREADS. For the DCE threads change the NULL to be pthread_mutexattr_default.

13 What's in Oct_tree.H?

Names

The Oct_tree class partitions 3-D space into quantized bins to enable quick searches that have the traditional time vs memory trade-off 13.2

13.1 template <class Type> union **Oct_data**

Oct_data a union of various pointer types

Members

Oct_data <Type> *
oct_ptr *Pointer to Oct_data<Type>*

A_a (Type) *Pointer to array containing pointers to items which all have identical quantized positions*

Oct_data <Type> *
oct [8] *Array of 8 pointers to Oct_data<Type>s*

inline Oct_data <Type> *
item (const int i) *Returns a pointer to the i'th item relative to the this pointer (not the usual *this)*

template <class R_a_array> inline void*
ref (R_a_array& ra)
Returns the reference value for this Block within the R_a

Oct_data a union of various pointer types. The is a helper union to encapsulate some of the most common uses of the Oct_tree Reference_Nc_arrays.

13.2

template <class Type, class PosType, class ObjType, class TagType> class **Oct_tree**

The Oct_tree class partitions 3-D space into quantized bins to enable quick searches that have the traditional time vs memory trade-off

Public Members

13.2.6		**Oct_tree** ()	*Default constructor*	93
		Oct_tree (Oct_tree& oct)	*Default copy constructor*	
13.2.7	void	**initialize** (ObjType* obj_, Vector<PosType>& lo_, Vector<PosType>& hi_, const PosType small, const PosType tol_, IntType hash_bits_, bool tagging_)	*Initialize the Oct_tree*	93
	void	**filled_sectors** (A_a(size_t)& sectors)	*Fills in Given array with a list of all hash table sectors that have at least one item in it*	
	size_t	**num_sectors** ()	*Returns the number of hash table sectors in the Oct_tree*	
	inline const Vector <PosType> &	**position** (Type *data)	*Return a reference to position vector calling user's pos_ref function*	
	inline TagType	**tag** (Type *data)	*Return the tag by calling user's tag() function*	
13.2.8	Type*	**find_near** (const Vector<IntType>& p, const TagType* tag_)	*Find a nearby item in the Oct_tree*	94
13.2.9	void	**remove** (Type* user_item)		

Remove a given item from the Oct_tree 94

13.2.10 inline void
insert (Type** at, Type* data)
Insert data item at supplied address 95

13.2.11 inline Type*
find (const Vector<PosType>& pos,
const TagType* tag_, Type**& dummy)
Find data item at given position and return an address if not found 95

inline Type*
query (const Vector<PosType>& pos,
const TagType* tag_)
Query to see if item exists in Oct_tree but do not modify the Oct_tree if item is not found

13.2.12 Type* **find** (const Vector<PosType>& pos,
const TagType* tag_, bool query,
Type**& dummy)
Find data item at given position and return an address if not found 95

void **report** (ostream& out)
Report Oct_tree diagnostics out

void **full_report** (ostream &out)
Full report Oct_tree diagnostics

virtual **~Oct_tree** () *Default destructor.*

The Oct_tree class partitions 3-D space into quantized bins to enable quick searches that have the traditional time vs memory trade-off. Given a position in 3-D space this class finds if something is in same bin. If tagging is enabled then a position and tag define uniqueness.

The Oct tree algorithm converts a floating point representation into an integer where each bit represents a level of an 8**n tree. 8**n comes about by dividing x,y,z space each into 2**n 1-D partitions. By MASKing the bits one can quickly determine the one of 8 slots to go down at a given level. A hash table is used to bypass the first hash_bits of levels. Items with identical positions but unique tags are legal.

13.2.6

Oct_tree ()

Default constructor

Default constructor. The user must call initialize before actually using the Oct_tree.

13.2.7

void **initialize** (ObjType* obj_, Vector<PosType>& lo_, Vector<PosType>& hi_, const PosType small, const PosType tol_, IntType hash_bits_, bool tagging_)

Initialize the Oct_tree

Initialize the Oct_tree.

```
The following variables supplied by the user determine the
behavior and memory overhead for the Oct_tree.

  1. ObjType pointer: Object used to extract position
     and tagging information
  2. Vector<PosType> lo_: Lowest spatial extreme the mesh
     will ever be
  4. Vector<PosType> hi_: Highest spatial extreme the mesh
     will ever be
  5. PosType small: Smallest spatial size that the
     Oct_tree should be able resolve
  6. PosType tol_: Spatial tolerance of the position data.
     Floating point representations of numbers within this
     range are considered to be identical.  The tol_
     parameter is in terms of small.  For example: 0.01 means
     the tol_ is 1 percent of the small parameter.  Numbers
```

```
   greater than 0.1 are mappped to 0.1 to help prevent
   ambiguity when tolerances are large
7. IntType hash_bits_: Number of hashed layers the Oct_tree
   is to use.  A higher number is faster but uses more
   memory.  The memory usage for the hash table is
   8**{hash_bits_+1} * sizeof(pointer) ,
8. bool tagging_:  boolean flag to determine if tagging
     will be used as a distiguishing feature for
     uniqueness
```

13.2.8

Type* **find_near** (const Vector<IntType>& p, const TagType* tag_)

Find a nearby item in the Oct_tree

Find a nearby item in the Oct_tree. This is not guaranteed to be the closest item. In general however, this member function will return an item within a Line Of Sight (LOS) of the position. A nearest neighbor may not be found if the number of hash bits is not set to 0. The user is issued a WARNING in this regard.

13.2.9

void **remove** (Type* user_item)

Remove a given item from the Oct_tree

Remove a given item from the Oct_tree. The item must have been inserted in the Oct_tree or error will result.

13.2.10

inline void **insert** (Type** at, Type* data)

Insert data item at supplied address

Insert data item at supplied address. This address must be obtained by using the find member functions. Range checks on done on this address to help prevent bugs but the tests are not full proof. The user must insert immediately after a find. If the Oct_tree is modified between a find and insert then the operation becomes ill-defined. Note that even a call to find can modify the Oct_tree.

13.2.11

inline Type* **find** (const Vector<PosType>& pos, const TagType* tag_, Type**& dummy)

Find data item at given position and return an address if not found

Find data item at given position and return an address if not found. The user must insert the item at the given address before the Oct_tree is modified or errors can result. Note that calling this routine modifies the Oct_tree if the position is not found.

13.2.12

Type* **find** (const Vector<PosType>& pos, const TagType* tag_, bool query, Type**& dummy)

Find data item at given position and return an address if not found

Find data item at given position and return an address if not found. The user must insert the item at the given address before the Oct_tree is modified or errors can result. Note that calling this routine modifies the Oct_tree if the position is not found and query is not set to true.

14 What's in Rb_tree.H

Names

The Rb_tree red black tree class is meant to hold large (many hundreds to billions) of data items 14.2
A variation of the Red Black tree.

14.1 template <class Key, class Data> class **Rb_data**

Rb_data class holds the data for a given entry in a Rb_tree

Public Members

inline **Rb_data** () *Default constructor.*

inline **Rb_data** (Rb_data<Key, Data>* lef, Rb_data<Key, Data>* righ, Rb_data<Key, Data>* par, Key& ky, Data& dat, Color col) *Constructs a Rb_tree given left, right, parent, key, data, and color*

inline Rb_data <Key, Data> *& **right** () *Returns a reference to the right Rb_data pointer*

inline Rb_data <Key, Data> *& **left** () *Returns a reference to the left Rb_data pointer*

inline Rb_data <Key, Data> *& **parent** () *Returns a reference to the parent Rb_data pointer*

inline Color&

color () — *Returns a reference to the color for this Rb_data*

inline Key&
key () — *Returns reference to key for this Rb_data*

inline Key **key_copy** () — *Returns a copy of the key*

inline Data&
data () — *Returns reference to data for this Rb_data*

inline Data
data_copy () — *Returns a copy ogf the data for this Rb_data*

inline Color
red () — *Returns the enum value of red*

inline Color
black () — *Returns the enum value of black*

14.1.1 inline void
relocate (Rb_data<Key, Data>*)
Free_list requires a relocate function 98

Protected Members

enum **Color** — *Enumeration that determines the color for this Rb_data*

Rb_data <Key, Data> *
left_ — *Points to a Rb_data item less than the current data*

Rb_data <Key, Data> *
right_ — *Points to a Rb_data item less than the current data*

Rb_data <Key, Data> *
parent_ — *Points to the parent in this red black binary tree*

Key **key_** — *The key belonging to this Rb_data*

Data **data_** — *The data belonging to this Rb_data*

Color **color_** — *The color of this Rb_data*

Rb_data class holds the data for a given entry in a Rb_tree. The class could be a nested class of Rb_tree but some compiliers don't like nested classes yet. Rb_data holds a left, right, and parent pointer, a key and a data item

14.1.1

inline void **relocate** (Rb_data<Key, Data>*)

Free_list requires a relocate function

Free_list requires a relocate function. This function is empty.

14.2

template <class Key, class Data> class **Rb_tree**

The Rb_tree red black tree class is meant to hold large (many hundreds to billions) of data items

Public Members

Rb_tree () *Default constructor*

Rb_tree (Index capacity_) *Constructor with capacity of size*

inline void **insert** (const Key& key, const Data& dat) *Inserts a data item into the red black tree overriding data if it already exists in the tree*

14.2.2 inline Data* **insert** (const Key& key) *Returns the address where data should be entered for this key* . . . 100

inline Data* **find** (const Key& key) *Returns pointer to data if item is in red black tree else 0*

inline void

remove (const Key& key)
Removes data item associated with provided key

inline Index
size () *Returns the number of items in this tree*

void **reinitialize** () *Reinitializes the red-black tree*

inline void
un_ordered (Index idx, Key& key, Data& dat)
Returns the i'th non-ordered key and data item in the tree

inline void
un_ordered_key (Index idx, Key& key)
Returns the i'th non-ordered data item in the tree

inline void
un_ordered_data (Index idx, Data& dat)
Returns the i'th non-ordered data item in the tree

The Rb_tree red black tree class is meant to hold large (many hundreds to billions) of data items. For smaller data sets use the Registry or Sorted array classes. There are significant memory and cpu overheads associated with using the Rb_tree.

While there are significant memory and cpu overheads, the insertion deletion, and find algorithms all assumptotically approach constant time for huge data sets.

The rotation, insertion and deletion member fuctions were adapted from c version of red black tree:

```
" By Thomas Niemann and is available on <A NAME="tex2html8"
  HREF="http://www.geocities.com/SoHo/2167/book.html">
  his algorithm collection webpages</A>.
  This code is not subject to copyright restrictions.
"
```

They have been modified to use Reference arrays. The functions have also been extended to work with data in a Registry fashion rather than just raw data.

Note that this implementation of the Rb_tree does not own the data but holds pointer to the data. It does own the keys.

14.2.2

inline Data* **insert** (const Key& key)

Returns the address where data should be entered for this key

Returns the address where data should be entered for this key. It is the callers responsibility to set the data up after being given this address.

15

template <class Type> class **Reference**

This creates a reference (&) to T data that is usually newed

Public Members

const Type*

data () const — *The const member functions are required by Array classes*

const Type&

data (const Index i) const

The const member functions are required by Array classes

This creates a reference (&) to T data that is usually newed. We must keep track of when the data moves and update the relocate_ref() member function. The sneekyness going on here is that we really have a pointer to an array of data but use data like it is a reference to a single data item. This amounts to one less dereferencing of a pointer and works for contiguous memory arrays. At present, there is a significant speed up over conventional pointer to array dereferencing. We might not need this class as compiliers mature. Use this class by inheriting this class and calling relocate_ref when data moves.

16

What's in Registry.H?

Names

Implements the Key_Data and Registry templatized classes.

16.1

template <class Key, class Data> class **Key_data**

The Key_data<Key,Data> class concatenates Key and Data into a single structure where the comparison operators are based soley on the Key

Public Members

Key_data () *Default constuctor*

Key_data (const Key& k_, const Data& d)
Construct a Key_data given key and data

Key_data (const Key& k_)
Constructor based on key alone. (Data is isgnored)

Key_data (const Key_data<Key, Data>& kd)
Copy constructor

inline bool

operator == (const Key_data<Key, Data>& rhs) const
== comparison operator based solely on key

inline bool
operator != (const Key_data<Key, Data>& rhs) const
!= comparison operator based solely on key

inline bool
operator > (const Key_data<Key, Data>& rhs) const
> comparison operator based solely on key

inline bool
operator >= (const Key_data<Key, Data>& rhs) const
>= comparison operator based solely on key

inline bool
operator < (const Key_data<Key, Data>& rhs) const
< comparison operator based solely on key

inline bool
operator <= (const Key_data<Key, Data>& rhs) const
<= comparison operator based solely on key

inline Key_data&
operator= (const Key_data& kd)
= equals operator

inline const Data&
dat () const *Return a reference to the data*

inline Data&
dat () *Return a reference to the data*

inline const Key&
key () const *Return a reference to the key*

inline Key&
key () *Return a reference to the key*

Protected Members

Key	**k**	*The key that unlocks the data*
Data	**data_**	*The data stored for a given key*

16.2

template <class Key, class Data> class **Registry**

The Registry class allows insertion and removal of data based on a key that is stored with the data

Public Members

Default constructor

Registry (const Index size = 0)
Construct a registry of given size

Registry (const Registry<Key, Data>& reg)
Copy constructor

inline Index
insert (const Key& k, const Data d)
Insert a key and its associated data item

inline void
remove (const Key& k)
Remove the key and its associated data item

inline Data
reg_data (const Key& k)
Return a copy of the data associated with the key

inline const Data&
ref_data (const Key& k)
Return a reference of the data associated with the key

16.2.2 inline Data
find_data (const Key& k)
Returns a copy of the data given a key 106

inline Data&

insert_data_ref (const Key& k)
This is not thread safe - be sure to lock down array before use

inline Data
operator[] (const Index i) const
Get the i'th data value in the array

inline Data
operator[] (const Index i)
Get the i'th data value in the array

16.2.3 inline Data&
operator() (const Index i)
Get a reference to the i'th data value in the array 106

inline Key **key** (const Index i) const
Return a copy of the i'th key

inline Key&
key_ref (const Index i) const
Return a reference to the i'th key

16.2.4 inline Index
find_position (const Key& k) const
Find the given key's position in the registry 106

16.2.5 inline Index
find_position (const Key& k, bool* exists)
Find the given key's position in the registry 107

inline Index
size () const *Returns the number of items in the registry*

inline void
capacity (const Index cap)
Sets the Registry's current capacity of the registry

inline Monitor*
monitor () *Returns the Array's monitor that contains keys and data*

void **report** () *Report out diagnostics about this registry*

~Registry () *Default destructor*

The Registry class allows insertion and removal of data based on a key that is stored with the data. Redundant keys are NOT reinserted!

16.2.2

inline Data **find_data** (const Key& k)

Returns a copy of the data given a key

Returns a copy of the data given a key. Returns 0 of the key is not valid. Calling this function makes sense only when data = 0 is meaningful.

16.2.3

inline Data& **operator()** (const Index i)

Get a reference to the i'th data value in the array

Get a reference to the i'th data value in the array. Not thread safe. Lock down the array before use.

16.2.4

inline Index **find_position** (const Key& k) const

Find the given key's position in the registry

Find the given key's position in the registry. This index is only guaranteed to correspond to given key until next insert or remove is called

16.2.5

inline Index **find_position** (const Key& k, bool* exists)

Find the given key's position in the registry

Find the given key's position in the registry. Sets boolean flag to true if it exists else false. This index is only guaranteed to correspond to given key until next insert or remove is called

17

enum **RetrieverTask**

Retriever task enumeration defininy types of tasks currently supported by the Retriever class

18

template <class Type, class SendData, class RtnData> class **Retriever**

The following class is a quick attempt at a Retriever class

Public Members

FuncPtr **func_ptr** (int task_type)
Returns the function pointer associated with given task type

18.5 **Retriever** (int task, Type *controller_, Communicator& communicator_, int comm_tag_, PmemFunc pmf_)
The Constructor sets up a Thread to perform a ThreadTask 111

18.6 static void* **single_data_sync** (void *untyped_ptr)
This member function is called when the Thread receives a request 111

inline void **no_op** ()
Does nothing but calm the ansi compiliers from warning about lack of usage of Retriever objects

~Retriever ()
Default destructor

The following class is a quick attempt at a Retriever class.

A general "user provided" member function is possible but not implemented. Instead a user must overload the retrieve function for a given class.

The intend is to overload the Constructor with different args to create threads that perform different types of Threading tasks. For example, a very general mechanism can be achieved by overloading retrieve function that accepts a Send and returning a Recv buffer. In other words, This class "encapsulates" the use of the Thread object.

A Retriever is something that manages a Thread. It constructs a Thread for a special purpose and destroys the Thread after completion. For certain Retriever tasks, an implicit barrier is setup in the destructor so that all procs in communicator must call the Retriever object's destructor before any of the other processor's destructors return to join with the thread of execution which spawned the request.

This class is very heavy weight and should be used when dozens or hundreds of messages are to be passed. This class is not meant for millions of requests. No buffering is done and the destructor could be expensive.

Restrictions:

1) Any member function of a class can be used but the member function must have exactly 2 arguments and be able to deal with requests from a foreign proccessor. The two arguments are sent data type and returned data type.

COMMENT BELOW is only relavant to SINGLE_DAT_SYNC

2) Although this allows any number of sends and receives without any user burden. There is an implicit barrier (destructor won't return until all destructors are called) which is Usually but not always what the user wants.

So a typical usage might be:

```
{  Retriever retriever(RetrieverTask,
                communicatior,
                controller,
                COMM_TAG,
                member_function called when retrieving);

   controller->member_function(retriever, data);
   .
   .
   .

   any user source code...
   .
   .

   controller->member_function(retriever, data);
   .
   .
   .
   any user source code...

}  <----.
        |
        |
        ' The closing braces implicitly calls destructor
          and all cleanup is done automatically.

User burden limited to constructor arguments. No Threading code
```

`required.`

18.5

Retriever (int task, Type *controller_, Communicator& communicator_, int comm_tag_, PmemFunc pmf_)

The Constructor sets up a Thread to perform a ThreadTask

The Constructor sets up a Thread to perform a ThreadTask. A RetrieverTask is a simple enumeration that corresponds to a static member function of this class which is the executable the Thread will actually run. The member function handles the management of the Thread. The actual data exchange is handled by the object passed to this constructor. The Communicator and comm_tag are used by the executable run by the Thread.

18.6

static void* **single_data_sync** (void *untyped_ptr)

This member function is called when the Thread receives a request

This member function is called when the Thread receives a request. Only a single data item is sent and received.

19

class **RunTime**

Times a program (or anything else you desire)

Times a program (or anything else you desire). Roughly accurate to the microsecond (depending on the underlying implimentation).

20

What's in Sparse_matrix.H?

Names

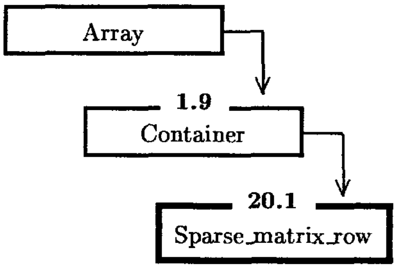
Declares and implements a simple Sparse_matrix class.

20.1

template <class Column, class Data> class **Sparse_matrix_row** : public Container<Key_data<Column, Data>, Sorted_Array<Key_data<Column, Data> > >

The Sparse_matrix_row is a decorated Sorted_Array container that provides a few additional services

Inheritance



Public Members

Sparse_matrix_row ()
Default constructor.

Sparse_matrix_row (const Row_Type& row)
Copy constructor.

inline Data&
operator) (const Column& col)
Returns a reference to the Data given a Column key

The Sparse_matrix_row is a decorated Sorted_Array container that provides a few additional services. This class is an interface class to rows of the Sparse_matrix class;

20.2

template <class Row, class Column, class Data> class **Sparse_matrix**

The Sparse_matrix is a simple convenience class for holding entries in a sparse matrix

Public Members

typedef Sparse_matrix_row <Column, Data>
Row_Type *Provides a convenient typedef for users to obtain a handle to the correct type of an row*

Sparse_matrix (Sstring filename, Index flush_int)
Construct sparse matrix setting up the filename and flush interval

inline void
check_access () *checks access useage and flushes matrix to file in access_count greater than flush_interval*

inline Data&
operator) (const Row& row, const Column& col)
Returns a reference to the data in the matrix entry for given row and column

void **flush_row** (Row& row, Row_Type& column_dat)

		Flush a row to disk writing row column data for each item in row
void	**flush_to_file** ()	*Forces a flush to file of all contents currently held in memory*

The Sparse_matrix is a simple convenience class for holding entries in a sparse matrix. The matrix is a two dimensional array of items.

This is not a general purpose sparse matrix class. The intended useage of this class is to incrementally fill a sparse matrix that, in general, may not fit in memory. The rows and column indices are templatized types with the most popular intended types of row, column being integral types.

For simplicity, we will optimize the use of row insertion and layout the memory accordingly. As a side-effect of this row overhead will be significantly higher (10 words or so). The column entry overhead will be one Column type and one Data type.

There are many bells and whistles that could be added to this class. At present this class is overly minimalistic.

There are no remove or deletion member functions since it is difficult to retrieve things flushed to disk efficiently. The user can get this effect by doing a -= of the current contents. Unfortunately, if the contents have been flushed to disk then the user has no easy way to know the final value of a (row,col) entry in the matrix.

```
Caveats: A performance assumption assumes that the number of
         column entries in a given row is relatively small
         (say typically less than a few hundred).  If the
         average number of column entries exceeds 100-200
         then a specialization of this class would be in order
         to bypass the Sorted array insertions which are
         of O(N*N) where N is the number of entries in a
         given row.

         This class makes no assumptions of Row and Column
         indicies being contiguous or even being of integral
         type.  What is required are the usual comparison
         functions and output >> operator for writing
         to disk.

         Future specializations of this class could take advantage
         contingous (or nearly so) indices.
```

21

#define **Sstring.H**

This Sstring class allows us to convert between various data types in addition to the usual string class utilities.

Sstring does not provide full functionality yet. The philosophy taken here is to build new functionality as needed.

Historical note: We tried using STL string classes several times but had difficulty extending it and a great deal of difficulty dealing with our limited debuggers. We rely heavily on Sstrings and the debuggers could not always parse through STL correctly.

22 class **Sstring**

The Sstring class allows us to convert between various data types in addition to the usual string class utilities

Public Members

friend inline ostream&
operator<< (ostream&, const Sstring&)
Allow ostream to be a friend of this class

friend istream&
operator>> (istream&, Sstring&)
Allow istream to be a friend of this class

friend Sstring
operator+ (const Sstring&, const Sstring&)
Allow global operator+ Sstring + String to be overloaded

friend Sstring
operator+ (const char*, const Sstring&)
Allow global operator+ char + String to be overloaded*

friend Sstring
operator+ (const Sstring&, const char*)
Allow global operator+ Sstring + const char to be overloaded*

Sstring () *Default constructor*

Sstring (const Sstring &s1)
Copy constructor

Sstring (const char *s1)
Specialized copy constructor for char's*

Sstring (const char *s1, size_t num)
Specialized copy constructor for fixed number of chars

Sstring (const short num)

Specialized copy constructor for short's

Sstring (const unsigned short num)
Specialized copy constructor for unsigned short's

Sstring (const int num)
Specialized copy constructor for ints's

Sstring (const unsigned int num)
Specialized copy constructor for unsigned int's

Sstring (const long num)
Specialized copy constructor for long's

Sstring (const unsigned long num)
Specialized copy constructor for unsigned long's

Sstring (const long long num)
Specialized copy constructor for long ints's

Sstring (const float num)
Specialized copy constructor for floats's

Sstring (const double num)
Specialized copy constructor for doubles's

Sstring (const bool val)
Specialized copy constructor for booleans's

22.3 **~Sstring** () *Default destructor* 121

size_t **size** () const *Return the string length in bytes*

SSTRING_OP_DEF(==)(!=)(>)(>=)(<)
(<=) (const unsigned int i)
== operator

Sstring& **operator** = (const Sstring &s1)
Assignment operator

Sstring& **operator** = (const char *s1)

*Specialized assignment operator for char *'s*

Sstring& **operator +=** (const Sstring &s1)
Concatenates s1 to end of Sstring.

Sstring& **operator +=** (const char *rhs)
Concatenates char to end of Sstring.*

operator const char* () const
User defined conversion from const char to Sstring*

operator char* () *User defined conversion from char* to Sstring*

operator bool () const
User defined conversion operator from Sstring to boolean

operator short () const
User defined conversion operator from Sstring to short

operator unsigned short () const
User defined conversion operator from Sstring to unsigned short

operator int () const
User defined conversion operator from Sstring to int

operator unsigned int () const
User defined conversion operator from Sstring to unsigned int

operator long () const
User defined conversion operator from Sstring to long

operator unsigned long () const
User defined conversion operator from Sstring to unsigned long

operator float () const
User defined conversion operator from Sstring to float

operator double () const

User defined conversion operator from Sstring to double

template <class Type>
operator Type* () const
User defined conversion operator from Sstring to a pointer of an abstract type

void **pack** (Send& buf) const
Pack this string into a Send buffer for communication

void **un_pack** (Recv& buf)
Unpack this string from a Recv buffer for communication

Protected Members

22.1 char* s *Our character string s* 122

22.2 void **copy** (const char *s1)
Copy the character string s1 to our string s 122

The Sstring class allows us to convert between various data types in addition to the usual string class utilities.

Sstring does not provide full functionality yet. The philosophy taken here is to build new functionality as needed. One of the main reasons for this class is to bypass dealing with STL string class in the debugger. Also we are transparent (syntatically and Memory layout) the C sstring class.

22.3

~Sstring ()

Default destructor

Default destructor. Frees memory if s != 0 Never inherit from this class. Note that this destructor is not virtual.

22.1

```
char* s
```

Our character string s

Our character string s. It is the first and only data item in this class so that its usage can be as close as possible with the normal sstring class. In particular, the memory layout should be identical so Sstring and string's can be used without conversions.

22.2

```
void copy (const char *s1)
```

Copy the character string s1 to our string s

Copy the character string s1 to our string s. We new a full copy. To help prevent memory leaks we force the caller of this routine to have set s to 0.

23

extern istream& **operator>>** (istream &i, Sstring &s1)

The following definitions are defined in Sstring

The following definitions are defined in Sstring.C to prevent multiple definition WARNINGs.

24

typedef long .int **Sbint**

Typedef abstracting what the Proc_dispenser, Block_index_mapper, and reader classes think a large Integral type is

Typedef abstracting what the Proc_dispenser, Block_index_mapper, and reader classes think a large Integral type is. We want these to be long long ints but all MPI implementations don't support 64 bits. (Noteably some IBM systems)

25

What's in Utilities.H?

Names

template <class T> void
Tcpy (T* p, const T* s, const int sz)
Not thread safe.

template <class T> void
Tset (T* p, T s, const int sz)
Not thread safe

template <class T> class
Alignment *Helper class for req_alignment GCC currently doesn't like nested structs that don't have explicit constructors*

template <class T> static size_t
req_alignment (const T&)
Find the required alignment of a given type

25.1 #define **Ntype** (Type_name)
The Ntype macro adds a "N_" prefix to the argument 126

25.2 #define **Type_to_enum_def** (AAA)
Now we will specialize concrete intrinsic types 126

25.3 template <class Type> static size_t
offset_positions (Type ** arr, size_t begin_off, size_t num_items, size_t* offs)
Given an Array of Type pointers this function fills in an array of offset positions for each item in array taking size and alignment into account 126

This file contains miscellaneous typdef's, enums, macros, generic algorithms, memory management stuff, etc.

25.1

#define **Ntype** (Type_name)

The Ntype macro adds a "N_" prefix to the argument

The Ntype macro adds a "N_" prefix to the argument. The following define converts a given typedef name to an enumeration. This macro is used in conjunction with the Intrinsic_enum enum above.

25.2

#define **Type_to_enum_def** (AAA)

Now we will specialize concrete intrinsic types

Now we will specialize concrete intrinsic types. Note that pointer and const variations are NOT taken into account here.

25.3

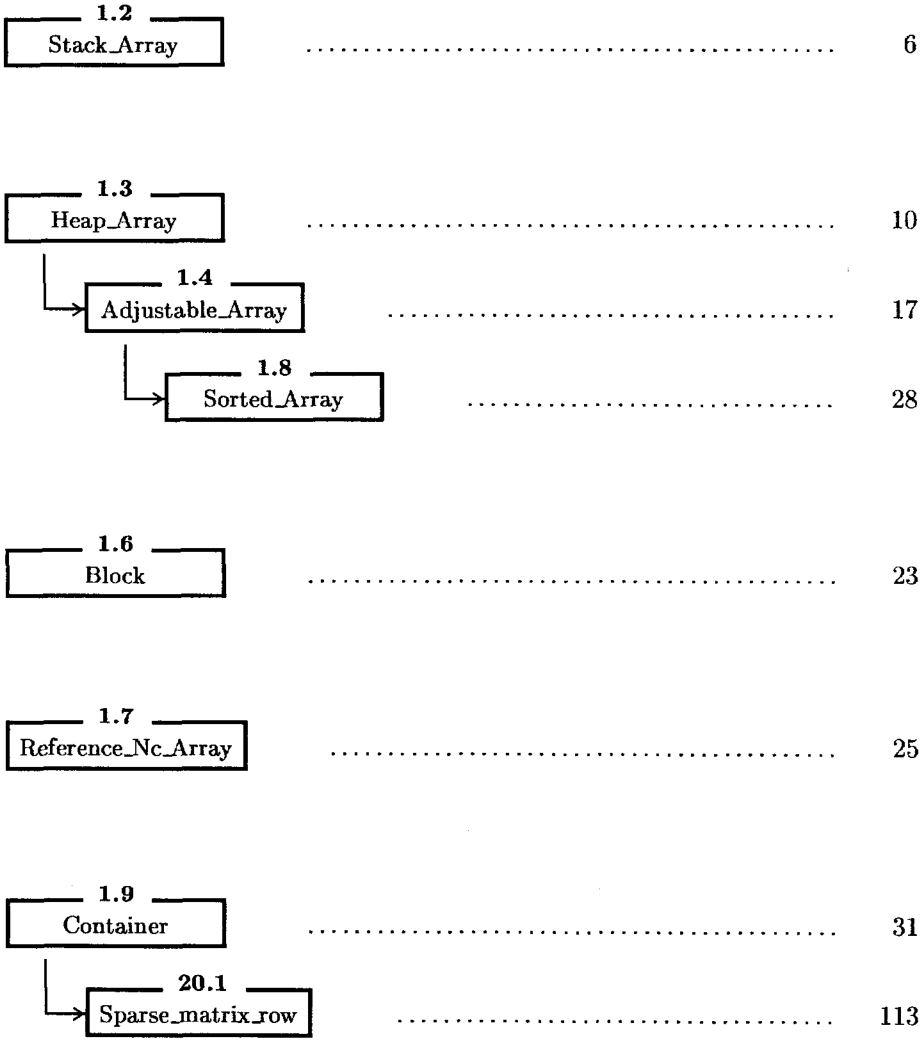
template <class Type> static size_t **offset_positions** (Type ** arr, size_t begin_off, size_t num_items, size_t* offs)

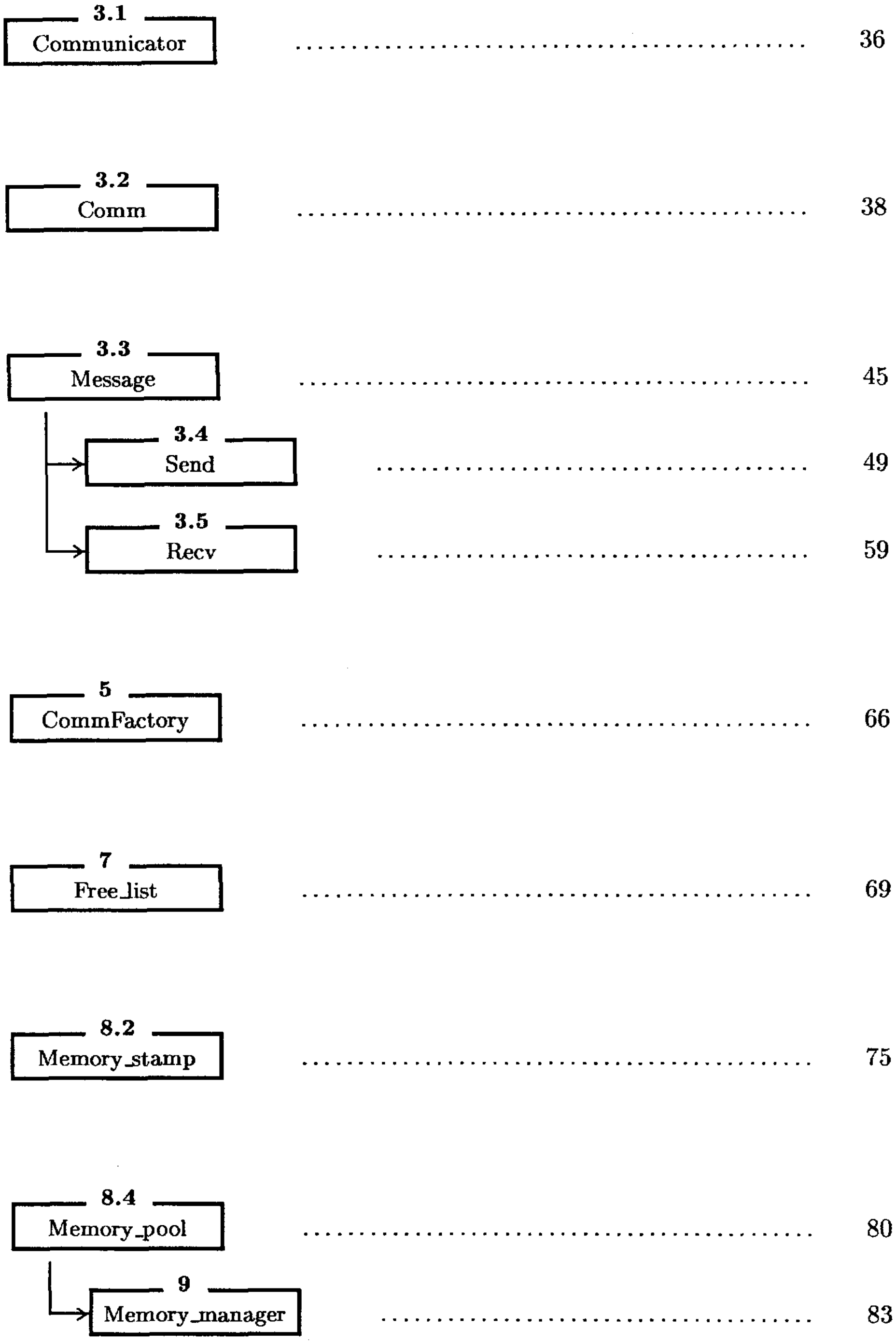
Given an Array of Type pointers this function fills in an array of offset positions for each item in array taking size and alignment into account

Given an Array of Type pointers this function fills in an array of offset positions for each item in array taking size and alignment into account. This function can begin with a beginning offset number of bytes. It is assumed that the 0 offset is aligned for strictest alignment of any data item in the array. This function returns the total aligned size for this array by padding the size on the strictest requiried alignment. This allows arrays of these structures to be packed contiguously.

The Type class must have size() and alignment() functions.

Class Graph





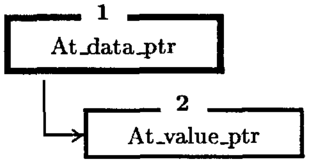
Contents

Entity-Attribute
Classes
Appendix B
Needs
covers

1 class **At_data_ptr**

Pointer to an At_data

Inheritance



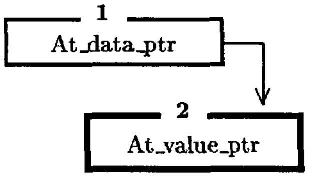
Pointer to an At_data. Comparisons between At_data_ptrs take place on the actual At_datas they represent.

2

template <class Type> class **At_value_ptr** : public At_data_ptr .

Pointer to an At_value

Inheritance



Pointer to an At_value. For conversions between At_data_ptr and At_value*

3

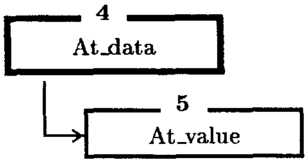
struct **Att_init**

Used to initialize At_data's for global Attributes, clones and copies

4 class **At_data**

Attribute can contain any number of data values which are species or local

Inheritance



Public Members

At_data (const Sstring & nam, Attribute * par)
Constructor called when making an At_data for an Attribute's clone or copy

virtual **˜At_data** () *Destructor to see that our subclasses are properly freed*

4.1 virtual int **compare** (const At_data& rhs) const
Compare two different At_data's 8

bool **operator<** (const At_data& rhs) const
Determine if this At_data is less than another

bool **operator<=** (const At_data& rhs) const
Determine if this At_data is less than or equal to another

bool **operator==** (const At_data& rhs) const
Determine if this At_data is equal to another

bool **operator!=** (const At_data& rhs) const
Determine if this At_data is not equal to another

bool **operator>=** (const At_data& rhs) const
Determine if this At_data is greater than or equal to another

bool **operator>** (const At_data& rhs) const

Determine if this At_data is greater than another

virtual At_data_ptr
copy () — *Make a copy of ourselves to be passed around like a bad rumour.*

virtual At_data_ptr
create (Sstring& init_str)
Create a new At_data using based off the provided string

virtual void
pack (Send&) const
Fill the given buffer to send the essence of an At_data.

virtual void
un_pack (Recv&) — *Set the At_data to whatever is contained in the Recv buffer.*

virtual void
unpack_local_data (Local*, Recv&)
Set the At_data value for this local to whatever is contained in the Recv buffer

Attribute can contain any number of data values which are species or local. At_data is the abstraction of this data.

4.1

virtual int **compare** (const At_data& rhs) const

Compare two different At_data's

Compare two different At_data's. First, we compare by name, if they're equal, we compare by type. This puts the order of At_data's by type to be: Computed > Common > Local The At_data's must be of the same Type of At_value.

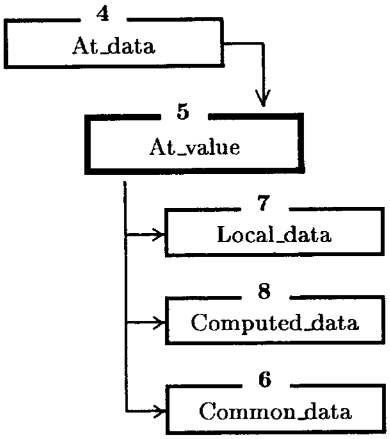
Return Value: s -1 if lhs<rhs, 0 if lhs==rhs, 1 if lhs>rhs

5

template <class T> class **At_value** : public At_data

An At_value is a basic wrapper for At_data

Inheritance



Public Members

	LocalArr	typedef void **(* FuncPtr)** (Entity *, Attribute *, void *ret)	*Signature of function used by computed data*	
		At_value (const Sstring & nam, Attribute * par)	*Basic constructor that creates an At_value of the proper type and value*	
		At_value ()	*Default constructor - this is used by the system (and internally within At_value) but should never be used explictly*	
	virtual	~**At_value** ()	*Run of the mill empty destructor*	
5.2	T&	**ref** (Local * loc)	*Get a constant reference to the data for the given local*	12
	const T&	**ref** (Local * loc) const		

			Get a constant reference to the data for the given local	
	T	**data** (Local * loc) const	*Get the data value for the given local*	
	void	**data** (const T& value)	*Set the data value to the given value - only valid for Common Data*	
	void	**data** (Local * loc, const T& value)	*Set the data value for the given entity*	
	void	**set_func** (const Sstring& str)	*Set the function that this uses if it's computed data*	
5.3	Sstring	**sstring** () const	*Get the value for this At_data as a string*	12
	Sstring	**sstring** (Local * loc) const	*Get the value for the given entity as a string*	
	void	**sstring_value** (const Sstring& value)	*Use this string to set the value for this At_data*	
	void	**sstring_value** (Local * loc, const Sstring& value)	*Use this string to set the value for this Entity*	
	void	**offset** (size_t offs)	*Set your offset to the given size*	
	size_t	**offset** () const	*Find out what offset is being used for this At_value.*	
	void	**array_size** (Index sz)	*Change the array size*	
	At_data_ptr	**copy** ()	*Make a copy of ourselves to be passed around like a bad rumour.*	
5.4	At_data_ptr	**create** (Sstring& init_str)	*Create a new At_data using based off the provided string*	12
	void	**pack** (Send& buf) const		

Fill the given buffer to send the essence of an At_data.

void **un_pack** (Recv& buf)
Set the At_value to whatever is contained in the Recv buffer.

void **unpack_local_data** (Local* loc, Recv& buf)
Set the At_data value for this local to whatever is contained in the Recv buffer

Protected Members

	T	**data_**	*real data for common, default data for local (when implemented)*	
	bool	**valid_offset**	*for local data*	
	size_t	**offset_**	*for local data*	
	FuncPtr	**fx**	*for calculated data*	
	Sstring	**fx_name**	*for calculated data*	
	Index	**arr_size**	*For local_array*	
	static Registry <Sstring, FuncPtr> *	**funcReg**	*for calculated data*	
	int	**owner**	*for communicating local data*	
	void	**data_value** (const T& value)	*Do the actual work of setting a value*	
	void	**func_value** (Sstring str)	*Do the actual work of setting a function for this computed data*	
	void	**all_data_values** (const T& value)	*Set the value of this and all copys of this common data*	
	void	**all_func_values** (const Sstring& fxname)	*Set the function pointer of this and all copys of this computed data*	
5.1	void	**set_copy** (At_value_ptr<T> copy_obj)	*Do the actual work of copying that a few functions need*	13

5.2

T& **ref** (Local * loc)

Get a constant reference to the data for the given local

Get a constant reference to the data for the given local. Note: this is neither thread nor comm safe. You have been warned.

5.3

Sstring **sstring** () const

Get the value for this At_data as a string

Get the value for this At_data as a string. At this level (no associated local*), we display the array size for Local arrays and the function name for calculated data.

5.4

At_data_ptr **create** (Sstring& init_str)

Create a new At_data using based off the provided string

Create a new At_data using based off the provided string. This string has the format "At_data-Type Type name value" where At_data-Type is 1 for common data, 2 for local data, 3 for calculated data or 4 for local array. Type should match our templated type, the name is just that and the value is the real value for common data, a default value for local data (not currently used), a function name (that's been registered) for calculated data or the size of the local array.

5.1

void **set_copy** (At_value_ptr<T> copy_obj)

Do the actual work of copying that a few functions need

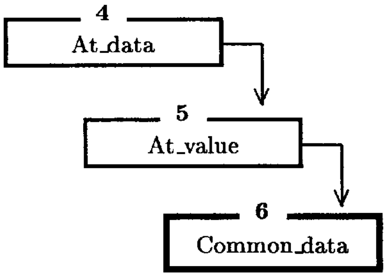
Do the actual work of copying that a few functions need. This sets the value of the copy based on the value of our member data.

6

template <class T> class **Common_data** : public At_value<T>

Common_data is is a special type of At_value

Inheritance



Public Members

6.1	T&	**ref** (Local *)	*This might not be thread-safe - data is*	14
	void	**data** (const T& value)	*Set the data value to the given value - only valid for Common Data*	
	void	**data** (Local * loc, const T& value)	*Set the data value for the given entity*	

6.1

T& **ref** (Local *)

This might not be thread-safe - data is

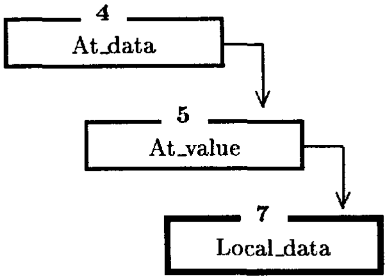
This might not be thread-safe - data is. This is also not safe for multi-processor operations.

7

template <class T> class **Local_data** : public At_value<T>

Local_data is a special type of At_value

Inheritance



Public Members

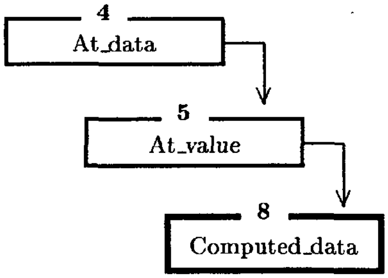
T&	**ref** (Local *loc)	*This might not be thread-safe - data is.*

8

template <class T> class **Computed_data** : public At_value<T>

Computed_data is a special type of At_value

Inheritance



9

class **Att_Notifiee**

Anyone who want's to be notified of new clones must inherit from this.

10

class **Attribute**

Attribute provides a standard mechanism to access data and it provides a tree representation of all data in the problem space

Public Members

static Att_init

global_ati — *The global At_data_init used to initialize the obj_ Attributes*

Attribute (Att_init& ati, const Sstring& nam = "", Attribute * par = NULL)
Attribute's basic constructor - this does nearly everything except cleaning the bathroom

10.2 Mutex* **lock** () — *Get a Mutex to lock down access to certain operations of all attributes* 21

static void

assign_gids () — *Assigns gids to all currently created Attributes of both type and kind ids*

int **proc_owner** (Gid gid)
Returns the processor rank number owns the given gid for this Attribute

10.3 static int **num_att_threshold** ()
Returns the number of Attributes (type_ids + kind_ids) threshold before the Species::update_att_threshold() mumber function is called 22

int **assign_did** (At_data_ptr dat)
Assign a data ID for At_datas.

10.4 void **copy_data** (A_a(At_data_ptr)&)

			Get a copy of the data for this Attribute	22
	Attribute*	**obtain_attribute** (const Sstring& clone_name, const Sstring& dat_name, const Sstring& dat_value)	*Get a clone of this species that has the given data set to the given value*	
	Attribute*	**obtain_attribute** (const Sstring& clone_name, const int dat_id, const Sstring& dat_value)	*Get a clone of this species that has the given data set to the given value*	
	virtual Attribute*	**clone** (const Sstring &, A_a(Sstring)&)	*Create a new clone of this Attribute type given an array of Sstrings to initialize its data values*	
	size_t	**size** () const	*Find the amount of local space this Attribute requires*	
	size_t	**alignment** () const	*Find the local alignment this Attribute requires*	
	void	**offset** (const size_t offs)	*Set the alignment of all the local data for this Attribute*	
	static Attribute*	**find_att** (const Sstring& name)	*Find an Attribute given its name*	
	static Attribute*	**find_att** (const int& kid)	*Find an Attribute given its ID*	
	At_data_ptr	**find_dat** (const int did)	*Find an At_data given its ID*	
10.5	virtual void	**init** ()	*Perform general initialization after main() has started*	22
10.6	virtual void			

finalize () — *Perform shutdown operations at the end of main()* 23

Protected Members

static Registry <Sstring, Attribute *>
att_nam_reg — *All attributes (except species specific), indexed by name*

static Registry <int, Attribute *>
att_id_reg — *All attributes (except species specific), indexed by ID*

static Att_Notifiee*
clone_notifiee — *Hook for notification of newly cloned Attributes*

static bool
at_most_one_ — *Determines if only one of this type of Att is allowed per Species*

static bool
comm_defining_ — *Determines if this Attribute should be transmitted across processors*

A_a (Attribute *) — *This only needs to be maintained for the root clone as long as all copies know who that is*

S_a (Species *) — *Array of species' that reference us;*

A_a (Attribute *) — *All clones of this type - only stored by the root_type*

S_a (Gid) — *Array of gid cutoffs to know who owns a given gid.*

A_a (At_data_ptr) — *ptrs to an kind's data*

Registry <int, At_data_ptr>
dat_reg — *All of the data items that this attribute has, indexed by did*

Registry <Sstring, At_data_ptr>
dat_nam_reg — *All of the data items that this attribute has, indexed by name*

Att_init **ati_** — *Used to initialize copies of the data*

Attribute* **gen_clone** (const Sstring &s,
A_a(Sstring)& at_val_strs)
This is called every time we need a new clone of a particular Attribute defined by an array of strings

template <class Self> Attribute*
gen_clone (const Sstring &name,
A_a(At_data_ptr)& dats, Self *)
This is called every time we need a new clone of a particular Attribute

10.1 template <class Self> Attribute*
species_copy (Self *, Species * species, int owner)
Species calls this for every Attribute in its list 23

void **assign_gid** () *Assign gids to all Entities that have this Attribute or one of this Attributes children*

Attribute provides a standard mechanism to access data and it provides a tree representation of all data in the problem space. Tricks: Clones itself to maintain tree structure across all processors. Copies itself to keep offset to local data for every species that uses it. Provides unique type_id and kind_id for easy comparison. Provides constraints to establish acceptable co-existance of two Attributes.

10.2

Mutex* **lock** ()

Get a Mutex to lock down access to certain operations of all attributes

Get a Mutex to lock down access to certain operations of all attributes. Currently this is just used for cloning to ensure that the clones are constructed in the same order on all processors.

10.3

static int num_att_threshold ()

Returns the number of Attributes (type_ids + kind_ids) threshold before the Species::update_att_threshold() mumber function is called

Returns the number of Attributes (type_ids + kind_ids) threshold before the Species::update_att_threshold() mumber function is called. This allows Species to be notified when the number of attributes has exceeded some threshold value.

10.4

void copy_data (A_a(At_data_ptr)&)

Get a copy of the data for this Attribute

Get a copy of the data for this Attribute. Note that this indirectly news memory for the At_datas, so they should be freed when you're done with them.

10.5

virtual void init ()

Perform general initialization after main() has started

Perform general initialization after main() has started. The user should always call this from main after the Comm object has been constructed.

10.6

virtual void **finalize** ()

Perform shutdown operations at the end of main()

Perform shutdown operations at the end of main(). The user should always call this if Comm is being used.

10.1

template <class Self> Attribute* **species_copy** (Self *, Species * species, int owner)

Species calls this for every Attribute in its list

Species calls this for every Attribute in its list. If the given Attribute has no local data, it returns itself. Otherwise, it will create an exact copy of itself and give that to the species.

11

(ent_p,att_type) extern Registry <Vector<int>, S_a(Attribute *)> **tag_to_atts**

*Global registry of Att_tag * to attributes lists*

12

void **att_map_reader** (char *filename)

a good old C style function!

13

const size_t **ENTITY_BLK_SZ**

At the moment we will hardwire the ENTITY_SIZE and ENTITY_MASK here

At the moment we will hardwire the ENTITY_SIZE and ENTITY_MASK here. The proper way is to call the Reference_Nc_Array containing the Entities but Entity is a lightweight class and can't afford the reference to the container. So we will use the mask.

14

#define **EF** (att_type)

Convenience define allows us the following syntax to call an Attribute's member functions:

Attribute mem_func mem_func_args — — — v v v EF(Topology)->up_ref(local, u, d)

15

class **Entity**

An Entity can be used to represent an object consisting of a collection of one or more Attributes

Public Members

15.1 inline **Entity** () *Default Constructor* 31

inline Species* **species** () *Returns the Species this Entity belongs to*

15.2 void **bind** (Species* new_species) *Binds an Entity to a species* 31

15.3 inline void **bind_local** (Local *loc) *Binds an Entity to a known Local* 31

15.4 inline Attribute* **operator()** (Attribute* req_att) const 32

15.5 inline Attribute* **operator()** (int att_id) const 32

15.6 inline void **remove** (Free_list<R_a(Entity), Entity>&, bool, bool) *Remove an entity* 32

15.7 inline void **relocate** (Entity *) *Relocate an entity to a new position* 32

15.8 inline void **add_att** (Attribute *new_att) *Adds an Attribute to an Entity* . 33

15.9 void **add_atts** (S_a(Attribute*)& new_atts) *Adds a list of Attributes to an Entity* 33

15.10 inline void

		remove_att (Attribute *old_att)	*Removes an Attribute from the Entity*	33
15.11	void	**remove_atts** (S_a(Attribute*)& old_atts)	*Removes a list of Attributes from the Entity*	34
	inline Gid	**gid** (Attribute* att_id)	*Returns the type_id Attribute specific gid for this Entity*	
	inline Gid	**gid** (int att_id)	*Returns the Attribute specific gid for this Entity*	
15.12	Gid	**geid** () const	*Returns the Global Entity ID* ...	34
	inline	**operator Local*** () const	*User defined conversion operator from Entity pointer to Local pointer so Entities and Locals can be used interchangeably*	
15.13	void	**pack** (Send& buf) const	*Pack this entity into a send buffer*	34
	static void	**backup** ()	*Produce a full backup of all Species data containing Entities*	
	static void	**swap_with_backup** ()	*Swap the current copy of all Species data with the backup copy*	
	static void	**remove_backup** ()	*Remove the backup copy of all Species data*	
	inline bool	**has_species** (Species* species_)	*Returns true if the Entity is of the provided Species*	
	inline bool			

has_species (const S_a(Species*)& species_list)
Returns true if the Entity's Species is one of the Species pointers in the provided Species list

15.14 inline bool
has_att (int att_id)
Returns true if the Attribute id provided is contained in the Entity's Attribute list 35

15.15 inline bool
has_att (Attribute *att)
Returns true if the Attribute provided is contained in the Entity's Attribute list 35

15.16 inline bool
has_att (const S_a(int)& attid_list)
Returns true if any of the Attribute ids provided in the Attribute list is contained in the Entity's Attribute list 35

15.17 inline bool
has_att (const S_a(Attribute*)& att_list)
Returns true if any of the Attributes provided in the Attribute list is contained in the Entity's Attribute list 36

15.18 inline **~Entity** () *Default Destructor* 36

Protected Members

Local* **local** *Pointer to Local which contains the actual data for this Entity*

An Entity can be used to represent an object consisting of a collection of one or more Attributes. It is a stable placeholder used to reference any data about the "Entity" it represents. Its data is organized by its Attributes. Entities that have the same set of Attributes are collectively managed by a Species. The actual data associated with an Entity lives in the Species data space. This class contains a single pointer that can access the data. The address of this pointer will not change without calling a relocate() member function. Derived types of this class should overload this function accordingly. By design this class has no virtual functions. If a derived type requires virtuals then Entity is probably not the appropriate class to use. A single virtual would incur 100Remember that an Entity consists of any number of Attributes and these Attributes can have all the virtuals one wants.

15.1

inline **Entity** ()

Default Constructor

Default Constructor. Initially points to nothing and must be bound before use.

15.2

void **bind** (Species* new_species)

Binds an Entity to a species

Binds an Entity to a species. This needs to be called before the Entity can be used. It may also be called to relocate the Entity to a new Species. The Entity will get its Local from the Species.

15.3

inline void **bind_local** (Local *loc)

Binds an Entity to a known Local

Binds an Entity to a known Local. Used by mutator threads.

15.4

inline Attribute* **operator()** (Attribute* req_att) const

Returns a pointer to the Species specific copy of the requested Attribute's type_id. If the Species is not associated with the given Attribute's type_id then NULL is returned. When the Attribute's id or type_id is known it is much faster to use the operator()(int id) member function.

15.5

inline Attribute* **operator()** (int att_id) const

Returns a pointer to the Species specific copy of the requested Attribute id. The id can be a type_id or kind_id. If the Species is not associated with the given Attribute id then NULL is returned.

15.6

inline void **remove** (Free_list<R_a(Entity), Entity>&, bool, bool)

Remove an entity

Remove an entity. If this leads to the removal of other entities then pack their addresses into the provided array for someone else to remove. Derived types should overload this member function accordingly.

15.7

inline void **relocate** (Entity *)

Relocate an entity to a new position

Relocate an entity to a new position. Derived types should overload this member function accordingly. Specifically, in this class we do not inform the Species or Local when we moved. If this is required we force the user to Derive from us and overload this function.

15.8

inline void **add_att** (Attribute *new_att)

Adds an Attribute to an Entity

Adds an Attribute to an Entity. The Entity will change Species.

15.9

void **add_atts** (S_a(Attribute*)& new_atts)

Adds a list of Attributes to an Entity

Adds a list of Attributes to an Entity. The Entity will change Species.

15.10

inline void **remove_att** (Attribute *old_att)

Removes an Attribute from the Entity

Removes an Attribute from the Entity. The Entity will change Species.

15.11

void **remove_atts** (S_a(Attribute*)& old_atts)

Removes a list of Attributes from the Entity

Removes a list of Attributes from the Entity. The Entity will change Species.

15.12

Gid **geid** () const

Returns the Global Entity ID

Returns the Global Entity ID. The Entity_container geid_sync() member function has previously setup the geid's.

15.13

void **pack** (Send& buf) const

Pack this entity into a send buffer

Pack this entity into a send buffer. That involves packing its geid, the list of defining attributes and their data.

15.14

inline bool **has_att** (int att_id)

Returns true if the Attribute id provided is contained in the Entity's Attribute list

Returns true if the Attribute id provided is contained in the Entity's Attribute list. The Entity Attribute list includes all parents up the Attribute tree. In particular, passing in the root Attribute id will always return true.

15.15

inline bool **has_att** (Attribute *att)

Returns true if the Attribute provided is contained in the Entity's Attribute list

Returns true if the Attribute provided is contained in the Entity's Attribute list. The Entity Attribute list includes all parents up the Attribute tree. In particular, passing in the root Attribute will always return true.

15.16

inline bool **has_att** (const S_a(int)& attid_list)

Returns true if any of the Attribute ids provided in the Attribute list is contained in the Entity's Attribute list

Returns true if any of the Attribute ids provided in the Attribute list is contained in the Entity's Attribute list. The Entity's Attribute list includes all parents up the Attribute tree. In particular, having the root Attribute id in the list of Attributes passed to this member function will always result in the return value being true.

15.17

inline bool **has_att** (const S_a(Attribute*)& att_list)

Returns true if any of the Attributes provided in the Attribute list is contained in the Entity's Attribute list

Returns true if any of the Attributes provided in the Attribute list is contained in the Entity's Attribute list. The Entity's Attribute list includes all parents up the Attribute tree. In particular, having the root Attribute in the list of Attributes passed to this member function will alway result in the return value being true.

15.18

inline ~**Entity** ()

Default Destructor

Default Destructor. This destructor is intentionally not virtual. Derived types are also meant to have non-virtual member functions and destructors.

16 class **Exists_on**

This class keeps track of the processor number that an entity resides on.

17

class **Ghost**

Tag an Entity as being owned by another processor.

18

template <class Type, class ArgType> class **Factory**

A general class to clone objects of a given type

Public Members

		Factory ()	*Default constructor*	
	virtual	**˜Factory** ()	*Default destructor*	
	bool	**find_arg** (const ArgType& arg, Index& pos)	*See if an argument exists and its position if it does*	
18.1	void	**update_arg** (const ArgType& old_arg, const ArgType& new_arg)	*Update an the given argument to a new value*	39
18.2	Type*	**clone** (ArgType& arg)	*Get a pointer to the unique instance of the given type defined by the given argument type*	40
	void	**all_clones** (A_a(Type*)& arr)	*Fills an array with all the clones we have manufactured to date*	

A general class to clone objects of a given type. Makes sure that there is only one instantiation of the given type for a given argument type.

18.1

void **update_arg** (const ArgType& old_arg, const ArgType& new_arg)

Update an the given argument to a new value

Update an the given argument to a new value. Called when the object that the given argument represents gets updated so that the argument and object continue to match.

18.2

Type* clone (ArgType& arg)

Get a pointer to the unique instance of the given type defined by the given argument type

Return Value: s Pointer to the unique instance

Parameters: The — argument that uniquely defines the requested instance

19

class **Token_obj**

This is a temporary comment for the Token_ob class

20

class **Filter**

This is a temporary comment for the Filter class

21

class **Initialize**

Initialization class is a place holder for items we want initialized before main is called and destroyed after main has ended

Public Members

		Initialize ()	*Default initialization*	
		Initialize (const Initialize&)	*Default copy constructor.*	
21.1	Initialize&	**operator=** (const Initialize& rhs)	*Equals operator*	43
	virtual	~**Initialize** ()	*Default Destructor. Hidden from everyone.*	

Initialization class is a place holder for items we want initialized before main is called and destroyed after main has ended.

Just for diagnostics. We want Memory_pool and Memory_manager to be first and last things constructed and deleted. This is required for accurate Memory_??? diagnotics

So we build a dummy class that hopefully gets contructed first and hence deleted last. In our destructor we actually call the diagnostic routines of the memory_manager

21.1

Initialize& **operator=** (const Initialize& rhs)

Equals operator

Equals operator. Shallow copy written prevent compilier warnings

22

class **Expression_operators**

Expression_operators class holds strings that name various operators

Public Members

Expression_operators ()
Default constructor.

22.1 inline void
insert (const Sstring& op_name, int precedence)
Inserts given operator name in registry and sets the precedence level for the operator 45

inline void
remove (const Sstring& op_name)
Removes given operator name from registry.

inline bool
is_operator (const Sstring& op_name)
Returns boolean true if given operator name is in the list of registered operator names

22.2 inline int **precedence** (const Sstring& op_name)
Returns the precedence level for given operator name 45

inline void
report () *Report out diagnostics about the Expression_operators*

~Expression_operators ()
Default destructor.

22.1

inline void **insert** (const Sstring& op_name, int precedence)

Inserts given operator name in registry and sets the precedence level for the operator

Inserts given operator name in registry and sets the precedence level for the operator. The precendence level must be greater than 0.

22.2

inline int **precedence** (const Sstring& op_name)

Returns the precedence level for given operator name

Returns the precedence level for given operator name. Returns -1 for the precedence level if operator is not in list of known operator names.

23

extern Expression_operators **CC_operators**

Holds the C++ Expression operators and precedences

24

const Sstring **parenthesis** ("()")

Const Sstring holding parenthesis characters

25

const Sstring **white_space** (” \t\n\r\f\v”)

Const Sstring holding white space characters

26

const Sstring **digit** ("0123456789")

Const Sstring holding decimal digits

27

const Sstring **xdigit** ("0123456789abcdefABCDEF")

Const Sstring holding hexidecimal digits

28

const Sstring **lower_alpha** ("abcdefghijklmnopqrstuvwxyz")

Const Sstring holding lower case alphabet characters

29

const Sstring **upper_alpha** ("ABCDEFGHIJKLMNOPQRSTUVWXYZ")

Const Sstring holding upper case alphabet characters

30

const Sstring **name_char** (”_” ”abcdefghijklmnopqrstuvwxyz” ”ABCDEFGHIJKLMNOPQRSTUVWXYZ” ”0123456789”)

Const Sstring holding all valid characters which can constitute a valid C++ variable name

31

const Sstring **operator_end** (" \t\n\r\f\v" "()" "0123456789" "abcdefghijklmnopqrstuvwxyz" "ABCDEFGHIJKLMNOPQRSTUVWXYZ")

Const Sstring holding set of all characters that can truncate a token of operator type

32

const int **MAX_TOKENS**

Maximum number of Tokens that can exist in a given Sstring

33

const int **MAX_TOKEN_LEN**

Maximum number of characters an indifidual token can be

34

enum **Token_kind**

Enumeration designating the types of tokens a string will be parsed into

35

bool **make_tokens** (Sstring inp, A_a(Sstring)& tokens, A_a(Token_kind)& token_kinds)

The make_tokens global function parses the given String into a array of tokens

The make_tokens global function parses the given String into a array of tokens. The function also returns an array of token types that correspond to each token token. There are currently six supported token types.

36

Sstring **convert_to_sstring** (A_a(Sstring)& array)

Concatenates an Array of Sstrings to a single sstring

37

bool **infix_to_postfix** (const A_a(Sstring)& infix, Expression_operators& exp_ops, A_a(Sstring)& postfix)

Converts an array of tokens in infix order to postfix order using the provided Expression_operators object to determine operators and precedences

Converts an array of tokens in infix order to postfix order using the provided Expression_operators object to determine operators and precedences.

The Expression_operator object contains a registry of operator Sstrings and their associated precedence level.

Any token not recognized by the Expression_operator is_operator() member function is assumed to be an operand. The left/right () parenthesis are recognized.

38

bool **infix_string_to_postfix** (const Sstring& inp, Expression_operators& exp_ops, A_a(Sstring)& postfix)

Converts a single infix Sstring to an array of postfix tokens using the provided Expression_operators object to determine operators and precedences

Converts a single infix Sstring to an array of postfix tokens using the provided Expression_operators object to determine operators and precedences.

The Expression_operator object contains a registry of operator Sstrings and their associated precedence level.

Any token not recognized by the Expression_operator is_operator() member function is assumed to be an operand. The left/right () parenthesis are recognized.

39

bool **infix_string_to_postfix** (const Sstring& inp, Expression_operators& exp_ops, Sstring& postfix)

Converts a single infix Sstring to a postfix Sstring using the provided Expression_operators object to determine operators and precedences

Converts a single infix Sstring to a postfix Sstring using the provided Expression_operators object to determine operators and precedences.

The Expression_operator object contains a registry of operator Sstrings and their associated precedence level.

Any token not recognized by the Expression_operator is_operator() member function is assumed to be an operand. The left/right () parenthesis are recognized.

40

const size_t **SPECIES_BLK_SZ**

At the moment we will hardwire the SPECIES_SIZE and SPECIES_MASK here

At the moment we will hardwire the SPECIES_SIZE and SPECIES_MASK here. The proper way is to call the Reference_Nc_Array of the local_data of a Species but this leads to a Local-Species circular depandancy. If the Array class or Species changes their MASK usage the following should be updated. There is an assert() test in the Species constructor to enforce this.

41

class **Local**

Local provides storage space for an Attribute's local data and provides a pointer back to the Entity who's data it holds so the Species it belongs to can find all of the Entitys it has

Public Members

	inline	**Local** ()	*Default constructor.*	
	inline	**Local** (Entity *ent)	*Constructs a Local filling in the provided Entity that owns this Local*	
	inline void	**entity** (Entity * ent)	*Sets the Entity that owns this Local.*	
	inline Entity*	**entity** ()	*Returns the entity that owns this Local.*	
	inline Species*	**species** ()	*Returns a pointer to the Species this Local belongs to*	
	Local&	**operator=** (const Local&)	*Equals operator*	
	void	**relocate** (Local *new_location)	*Relocate the current Local to the specified new_location*	
41.2	inline	**~Local** ()	*Default destructor*	65

41.2

inline **~Local** ()

Default destructor

Default destructor. The destructor does nothing and is intentionally non-virtual.

42

class **Outer_limits**

Features common to outer surface/outer layer of a processor. This is convenience Attribute that can be added to Entities.

43

class **Att_info**

The Att_info class is a helper class that holds Attribute specific infomation relative to a particular Species

Protected Members

	friend class	**Species**	*Allow Species as a friend class for convenient access to all of the protected data items*	
	Attribute*	**orig**	*Pointer to either a root clone or an obj_??? Attribute*	
	Attribute*	**ours**	*Pointer to pur Species specific copy of an Attribute.*	
	Att_info*	**next**	*Pointer to the next Att_info for Attributes an id in common*	
	Gid	**base_gid**	*Holds The Species specific beginning index of the global id.*	
	size_t	**att_local_size**	*Holds the total local byte size of this attribute*	
	size_t	**offset**	*Holds the number of bytes offset from the beginning of the Local data that this Attribute's data begins at*	
	size_t	**next_att_offset**	*Offset from beginning of Local data where the next attribute begins its data*	
		Att_info ()	*Default constructor.*	
		Att_info (Attribute *orig_, Attribute *ours_, Att_info *next_, Gid base_gid_, size_t size_, size_t offset_, size_t next_att_offset_)	*Constructs an Att_info with given data.*	
	Att_info&	**operator=** (const Att_info& rhs)	*Equals operator*	
43.1	virtual	**~Att_info** ()	*Default destructor*	68

43.1

virtual ~Att_info ()

Default destructor

Default destructor. Will delete all Att_info objects pointed to by the next pointer.

44

class **Species_info**

The Species_info class is a helper class to Species that holds Species - Species memory relocation information

Public Members

Species_info ()	*Default constructor.*
Species_info (const A_a(size_t)& our_off, const A_a(size_t)& their_off, const A_a(size_t)& seg_sizes_)	*Constructs a Species_info object given our & their offsets and memory segment sizes*
Species_info (const Species_info& rhs)	*Default Copy constructor.*
Species_info& **operator=** (const Species_info& rhs)	*Equals operator.*
~**Species_info** ()	*Default Destructor.*

Protected Members

A_a (size_t)	*Array of offsets into our Local data*
A_a (size_t)	*Array of offsets into the Species we are copying data our Local data from*
A_a (size_t)	*Array of memory segment sizes*

45

class **Species**

Class Species manages a collection of Entity Locals (local data for an Entity) that each have an identical set of Attributes

Public Members

Species (const S_a(Attribute*)& att_args)
Constructs a Species given a set of Attributes that define the Species

inline Local*
insert (Entity* entity)
Insert a new Local into the local data array and initialize the local with the Entity which owns it

inline Local*
insert (Entity* entity, Local* old_local,
Species* old_species)
Insert a new Local into local data array from an another Species

45.10 Species_info*
make_species_info (Species* old_species)
Create and fill a new Species info for moving data from that Species to this Species 74

45.11 inline Attribute*
operator) (Attribute * req_att) const
Returns the our Species specific Attribute for the given Attribute pointer if one exists 74

45.12 inline Attribute*
operator) (const int req_id) const
Returns the our Species specific Attribute for the given Attribute id if one exists 74

inline size_t

comm_defining_size () const
Returns the local data size of data that has been tagged as comm_defining for this Species

inline void
comm_defining_kids (A_a(int)& kids)
Fills in an array of the Attribute id's that are comm_defining for this Species

inline size_t
local_size () const *Returns the local data size for this Species.*

inline S_a **(Attribute *)** () const
Returns an array of the original Attributes used to construct this Species

inline A_a **(Attribute *)** () const
Returns a copy of the Species specific Array of Attributes

static inline Species*
clone (S_a(Attribute*)& args)
Returns a pointer to a cloned Species having the given set of Attributes

static inline Species*
clone (S_a(int)& args)
Returns a pointer to a cloned Species having the given set of Attributes

static inline void
all_species (A_a(Species *)&arr)
Fills an array of pointers to all Species currently in existance

45.13 void inline
all_atts (int id, A_a(Attribute *)& att_list)
Fills a list of all the Attributes of a given id contained in this Species

75

inline Index
entity_count () *Returns the total number of Entities that belong to this species*

inline Entity*

entity (const Index i)
Returns the i'th Entity for this Species

inline void
remove (Local *old)
Removes an Entity's local data

inline void
backup () *Produces a full backup of all Local_data for this Species*

inline void
swap_with_backup ()
Swaps the current Species data with the backup copy

inline void
remove_backup ()
Removes the backup copy of this Species

45.14 inline void
lock () *Locks down the Species so no insertions, removal, or relocations can occur* 75

45.15 inline void
un_lock () *Unlocks the Species so insertions, removals, and relocations can occur* 75

inline bool
comm_species () *Returns boolean true if this Species has either a Ghost or an Exists on Attribute*

inline bool
ghost () *Returns boolean true if this Species has a Ghost Attribute*

inline bool
exists_on () *Returns boolean true if this Species has a Exists_on Attribute*

inline Gid **gid** (Local *local, int att_id)
Returns the Gid for the given Local's Attribute id

45.16 inline Gid **gid** (Local *local, Attribute* att)

Returns the Gid for the given Local's Attribute pointer 76

inline Species*
add_atts (S_a(Attribute *)& new_atts)
Returns a pointer to the Species that has the given additional Attributes

inline Species*
remove_atts (S_a(Attribute *)& old_atts)
Returns a pointer to the Species that has the given Attributes removed

void **report** () *Prints out a report for this Species.*

~Species () *Default Destructor*

Protected Members

inline void
species_relocate (size_t new_loc, size_t old_loc, const Species_info& species_info)
Copies data from old location to new location using the provided Species_info object

45.8 static void
update_atts_threshold (int att_threshold)
Updates all att_idx_arrs to correct size 76

45.9 void **assign_base_gid** (int id, Gid base_gid_)
Assigns a base_gid for a given Attribute id 76

Class Species manages a collection of Entity Locals (local data for an Entity) that each have an identical set of Attributes. Each Entity then has the same amount of data associated with it. Species class also helps Entity - Attribute relationships that are common to this particular Species.

45.10

Species_info* **make_species_info** (Species* old_species)

Create and fill a new Species info for moving data from that Species to this Species

Create and fill a new Species info for moving data from that Species to this Species. The Local data for each attribute with identical kind ids is copyied when an Entity moves from one Species to another. If kind ids are different but type ids are the same then we copy data only if the at_most_one() constraint is set true for the Attribute under consideration. Of special note is the possibility of data changing sizes even if the type_id is the same. If this is the case only the minimum of the two data sizes is copyied (from the beginning data offset). It is the user's responsibility to fill in any remaining data and ensure that any desired data to be copied is resides at the beginning data space.

45.11

inline Attribute* **operator)** (Attribute * req_att) const

Returns the our Species specific Attribute for the given Attribute pointer if one exists

Returns the our Species specific Attribute for the given Attribute pointer if one exists. Null is returned if non exists.

45.12

inline Attribute* **operator)** (const int req_id) const

Returns the our Species specific Attribute for the given Attribute id if one exists

Returns the our Species specific Attribute for the given Attribute id if one exists. Null is returned if none exists.

45.13

void inline **all_atts** (int id, A_a(Attribute *)& att_list)

Fills a list of all the Attributes of a given id contained in this Species

Fills a list of all the Attributes of a given id contained in this Species. The Attribute pointers are pointers to Species specific copyies.

45.14

inline void **lock** ()

Locks down the Species so no insertions, removal, or relocations can occur

Locks down the Species so no insertions, removal, or relocations can occur. This lock is not checked. The user has agreed to not call insert, remove, or relocate functions while the lock is in place. Performance issues prevent the "real" locking down if the Species. Modifing the Species while locked down will, in general, only be detected in DEBUG mode.

45.15

inline void **un_lock** ()

Unlocks the Species so insertions, removals, and relocations can occur

Unlocks the Species so insertions, removals, and relocations can occur. The free_list automatic garbage_collection is turned on and free_list emptied at this time.

45.16

```
inline Gid gid (Local *local, Attribute* att)
```

Returns the Gid for the given Local's Attribute pointer

Returns the Gid for the given Local's Attribute pointer. The type_id is used to determine exact Attribute.

45.8

```
static void update_atts_threshold (int att_threshold)
```

Updates all att_idx_arrs to correct size

Updates all att_idx_arrs to correct size. This function is called by Attribute when the number of Attributes has grown beyond some threshold.

45.9

```
void assign_base_gid (int id, Gid base_gid_)
```

Assigns a base_gid for a given Attribute id

Assigns a base_gid for a given Attribute id. This base Gid is used to calculate a Local's gid for a given attribute. Calling this member function locks down the Species. The user agrees to not insert, remove or relocate any Local data because this would invalidate the gid calculations.

Class Graph

Mesh Classes
Appendix C

Contents

1 class **Boundary_condition**

Features common to boundary conditions.

2

class **Truncation_boundary**

Names

inline void **truncation_id** (int trunc_id)
Fills in a truncation_id for this Attribute.

inline int **truncation_id** () *Returns a copy of the Truncation id for this Attribute.*

Features common to the truncation of the problem space.

3

class **Absorbing_boundary_condition**

Features common to the Absorbing boundary conditions. For now this is set as a Truncation boundary but there are Absorbing boundary conditions that are not truncation boundaries in the strictest sense of the word.

4

class **Perfect_electrical_conductor**

Features common to the perfect electrical conductors.

5

class **Cell**

Features common to Cells

6

class **Edge**

Features common to Edges

7

class **Face**

Features common to Faces

8

class **Geometry**

Tag the non-Topology attributes of a mesh.

9 class **Dual**

Tag that an element is a member of the dual mesh.

10

Mesh.H

Names

This file just defines the Mesh class, which is the highest level abstraction in the Tiger purview. Above here, it's all physics (for the time being).

10.1

extern Mesh **obj_Mesh**

Mesh really should be a abstract base class

Names

void	**(*order_list)** (A_a(int)&)	*Function to use to order our list of neighbors in global_build()*
bool	**initialized_**	*Boolean flag signaling whether initialize() member function has been called*
static int	**num_parts_**	*Holds the number of Mesh parts that constitute the entire Mesh*
static Entity_container <Mesh_entity>	**ec**	*This Entity Container holds all the Mesh Entities used by this processor*
static Registry <Sstring, Mesh *>		

		mesh_types_reg_	*Registry with the key being the Derived Class Sstring name and the Data being a pointer to the corresponding obj_Derived_Mesh_class*	
	static Oct_tree <Mesh_entity, Mesh_position_type, Node, TagType>	**oct_**	*The local Oct_tree for this processor*	
	static Oct_tree <Mesh_entity, Mesh_position_type, Node, TagType>	**g_oct**	*The global Oct_tree for all processors*	
	static Vector <Mesh_position_type>	**lo**	*Contains local minimum extremes of the Mesh*	
	static Vector <Mesh_position_type>	**hi**	*Contains the local maximum extremes of the Mesh*	
	static Vector <Mesh_position_type>	**glo**	*Contains global minimum extremes of the Mesh*	
	static Vector <Mesh_position_type>	**ghi**	*Contains the global maximum extremes of the Mesh*	
	static	**A_a** (Mesh *)	*Contains an array of pointers to Mesh part projects belonging to this Mesh*	
10.1.1	virtual Mesh*	**clone** (const Sstring&)	*This member function serves a signature for Derived classes* ...	18
	TagType	**part_id** (Mesh *)	*Given a pointer th a Mesh part, this member function returns that part's id number*	
10.1.2		**Mesh** ()	*Default Constructor*	15
		Mesh (const Sstring& meshparts_file)	*Constructs a Mesh from the given file of Mesh parts*	
10.1.3	virtual void	**initialize** ()	*Initialize the Mesh part*	16
10.1.4	void	**initialize_oct** ()	*Initializizes the Oct_tree*	16

template <class Derived_Type> inline Mesh*
gen_clone (const Sstring& mesh_file,
Derived_Type*)
Given Mesh type, reader format, and mesh filename Sstrings This member function returns a pointer an initialized mesh object

10.1.5 virtual void
local_build () *Builds the Mesh into the Entity Container* 16

10.1.6 void **geid_sync** () *Syncs the Entities by providing a unique global Entity identification number (geid) accross all processors* 17

10.1.7 inline bool
geid_synced () *Returns boolean answering the question of whether the geid_sync() member function has been called* 17

10.1.8 void **global_build** () *Converts a series of local Meshes for each processor into a single unified Mesh* 17

virtual void
close () *Closes the reader file*

inline size_t
size () *Returns the number of Mesh_Entities currently in the Mesh*

inline void
extend_bounds (const
Vector<Mesh_position_type>& pos,
Vector<Mesh_position_type>& lo_,
Vector<Mesh_position_type>& hi_)
Given a position, along with lo and high extremes, this member function modifies the extremes if position is outside present bounds

inline void

extend_bounds (const Vector<Mesh_position_type>& pos)
Given a position, this member function modifies the Mesh lo and hi extremes if position is outside present bounds

void **limits** (Vector<Mesh_position_type>& vlo, Vector<Mesh_position_type>& vhi)
Fills in the local lo and high extremes of Mesh

inline Index **index** (Mesh_entity *e)
Returns the position index into the Entity_container

void **finalize** ()
This member function calls the Entity_container to finalize all required communication

~Mesh ()
Default Destructor

Mesh really should be a abstract base class. However, some compiliers need a little help getting everything initialized. Also, it is convenient to have the obj_Mesh for access rather than requiring the user to know the mesh type at this level. As compilier mature, if obj_Mesh is removed below then remember to make the empty member functions above pure virtuals.

10.1.2

Mesh ()

Default Constructor

Default Constructor. Sets initialization boolean to false. The user must call initialize() before using this Mesh object.

10.1.3

virtual void **initialize** ()

Initialize the Mesh part

Initialize the Mesh part. Derived types must overload this member function. The nodes of the Mesh file part are read in during this initialization phase. Also, limits of the mesh part are determined. Mesh's initialize does nothing, derived types do the work.

10.1.4

void **initialize_oct** ()

Initializizes the Oct_tree

Initializizes the Oct_tree. TODO: Initialization of the Oct_tree should allow user-defined parameters to be passed in here. For now minimum bin size, tolerance, and hash table size are hard wired to reasonable values in this member function.

10.1.5

virtual void **local_build** ()

Builds the Mesh into the Entity Container

Builds the Mesh into the Entity Container. Each derived specialization of this class must write this function. Mesh's local_build does nothing, derived types do the work.

10.1.6

void **geid_sync** ()

Syncs the Entities by providing a unique global Entity identification number (geid) accross all processors

Syncs the Entities by providing a unique global Entity identification number (geid) accross all processors. In general, geid's will not be contiguous. After calling this member function no totally new Entities can be created. New Entities created on a given processor must have existed on some other processor before calling this geid_sync function.

10.1.7

inline bool **geid_synced** ()

Returns boolean answering the question of whether the geid_sync() member function has been called

Returns boolean answering the question of whether the geid_sync() member function has been called. If geid_synced() == true then use of geids valid.

10.1.8

void **global_build** ()

Converts a series of local Meshes for each processor into a single unified Mesh

Converts a series of local Meshes for each processor into a single unified Mesh.

10.1.1

virtual Mesh* **clone** (const Sstring&)

This member function serves a signature for Derived classes

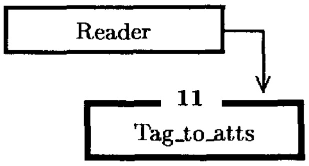
This member function serves a signature for Derived classes. Derived class must overload this function. Mesh's clone function just returns the this pointer. Derived classes will do the work of actually cloning a Mesh part object.

11

template <class Reader> class **Tag_to_atts** : public Reader

The Tag_to_atts class is a decorator class for Mesh_reader classes

Inheritance



Public Members

11.5 inline **Mesh_entity** () *Default constructor* 22

inline void
up (A_a(Mesh_entity*)& ups)
Fills the given array with the up pointers for this Entity

inline void
dn (A_a(Mesh_entity*)& dns)
Fills the given array with the dn pointers for this Entity

inline Mesh_entity**
up_ref () *Returns reference to the ups for this Entity*

inline Mesh_entity**
up_ref (Index& nup)
Fills in the number of ups and returns reference to the ups

inline Mesh_entity**
up_ref (Index& nups, Index& ndns)
Fills in the number of ups and dns returns and reference to the ups & dns

inline Mesh_entity**

dn_ref () *Returns a reference to the dns for this Entity*

inline Mesh_entity**
dn_ref (Index& ndn)
Fills in the number of dns and returns reference to the dns

void **add_up** (Mesh_entity* to_add)
Adds an up to this Entity

void **add_dn** (Mesh_entity* to_add)
Adds a down to this Entity

11.6 inline void
remove (Free_list<R_a(Mesh_entity),
Mesh_entity>& free_list, bool uflag,
bool dflag)
Removes an entity 22

void **remove_up** (Mesh_entity* to_sub,
Free_list<R_a(Mesh_entity),
Mesh_entity>& free_list)
Removes the given up from this Entity

void **remove_dn** (Mesh_entity* to_sub,
Free_list<R_a(Mesh_entity),
Mesh_entity>& free_list)
Removes the given dn from this Entity

11.7 inline void
relocate (Mesh_entity *e)
Relocates an entity to a new position 23

inline void
replace_up (Mesh_entity* old_address,
Mesh_entity* new_address)
Replaces up with old address with the up's new address

inline void
replace_dn (Mesh_entity* old_address,
Mesh_entity* new_address)
Replaces dn with old address with the dn's new address

11.8 void **report** () *Prints a report for this entity* ... 23

11.9 void **report** (int level) *Report of entity's ups and dns recursively from level on down* ... 23

inline **operator Sstring** () const
Letting Sstring know that we don't want anything to do with it

inline bool
has_up (Mesh_entity *entity)
Returns boolean true if any of this Entity's ups is identical to the provided Entity

inline bool
has_up (const S_a(Mesh_entity*)& entity_list)
Returns boolean true if the Entity has an up identical to any of the Entities in the provided list of Entities

inline bool
has_dn (Mesh_entity *entity)
Returns boolean true if any of this Entity's dns is identical to the provided Entity

inline bool
has_dn (const S_a(Mesh_entity*)& entity_list)
Returns boolean true if the Entity has an dn identical to any of the Entities in the provided list of Entities

11.10 inline **~Mesh_entity** () *Default Constructor* 24

inline void
next_node (Gid& id, Vector<double>& pos)
Reads the next node record returning the node id and node position

11.11 inline void
next_cell (Gid& id, S_a(Attribute *)& 'atts,
A_a(Gid)& nodes)
Reads the next cell record and uses tagging to obtain a set of attributes that will accompany the cell 24

11.12 inline void

The Tag_to_atts class is a decorator class for Mesh_reader classes. Tag_to_atts abstracts the tagging of nodes, cells, and specials by converting from a Mesh_tag to a set of Attributes.

11.5

inline **Mesh_entity** ()

Default constructor

Default constructor. Intentionally does nothing

11.6

inline void **remove** (Free_list<R_a(Mesh_entity), Mesh_entity>& free_list, bool uflag, bool dflag)

Removes an entity

Removes an entity. Passes a freelist along so everyone called can add to list all other Entities that should now be removed as a result of this Entity's removal.

11.7

inline void **relocate** (Mesh_entity *e)

Relocates an entity to a new position

Relocates an entity to a new position. The ups and dns for this entity are called and the up&dn references pointing back to this Entity are updated with new location.

11.8

void **report** ()

Prints a report for this entity

Prints a report for this entity. Up, dn, and Species information are reported.

11.9

void **report** (int level)

Report of entity's ups and dns recursively from level on down

Report of entity's ups and dns recursively from level on down. No Species specific information is provided.

11.10

inline **~Mesh_entity** ()

Default Constructor

Default Constructor. This destructor is intentionally not virtual. Derived types are also meant to have non-virtual member functions and destructors.

11.11

inline void **next_cell** (Gid& id, S_a(Attribute *)& atts, A_a(Gid)& nodes)

Reads the next cell record and uses tagging to obtain a set of attributes that will accompany the cell

Reads the next cell record and uses tagging to obtain a set of attributes that will accompany the cell. Cells are defined by their nodes.

11.12

inline void **next_special** (Gid& id, S_a(Attribute *)& atts, A_a(Gid)& nodes)

Reads the next special record and uses tagging to obtain a set of attributes that will accompany the special

Reads the next special record and uses tagging to obtain a set of attributes that will accompany the special. Specials are defined by their nodes.

12

typedef double **Mesh_position_type**

We don't want to templatize here but we want to abstract it anyway

We don't want to templatize here but we want to abstract it anyway. Perhaps this belongs in Utilities.H

13

inline void **cartesian_position** (Entity *ent, Attribute *att, void *ret)

Fills ret with a pointer to a valid Vector position for cartesian Node Attributes

14

class Node

Names

void **pos** (Mesh_entity *e,
Vector<Mesh_position_type>& v)
Fills in a position Vector for a given Mesh_entity

Vector <Mesh_position_type>
pos (Mesh_entity *e) const
Returns a copy of the position for given entity.

const Vector <Mesh_position_type> &
pos_ref (Mesh_entity *e) const
Returns a reference to position data for a given entity.

void **pos** (Mesh_entity *e,
const Vector<Mesh_position_type>& v)
Fills a const reference to position data

void **pos** (Mesh_entity *e,
Vector<Mesh_position_type>*& v)
Fills a pointer to position data

void **tag** (const TagType& t)
Sets the tag for this Attribute

TagType **tag** () *Returns a copy of the tag for this attribute*

void **mesh_part_file** (const Sstring& mesh_file)
Sets the mesh_part_file for this Attribute.

Sstring **mesh_part_file** () *Returns a copy of the mesh part filename for this attribute*

void **Node::cartesian_pos** (Mesh_entity *, Vector<Mesh_position_type>*)
Required for calling index function of obj_Mesh

Features common to Nodes

15

```
void processor_allocator (int num_procs, const
                          A_a(double)& weighted_totals,
                          A_a(A_a(int))& part_to_proc,
                          A_a(A_a(int))& proc_to_part)
```

This global function distributes the given number of processors accross each of the parts

This global function distributes the given number of processors accross each of the parts. The implementation is "reasonable" but not optimal.

16

void **block_partitioner** (int num_blocks, const Vector<Ubint>& orig_block, const Vector<Ubint>& orig_offsets, A_a(Vector<Ubint>)& block_starts, A_a(Vector<Ubint>)& block_sizes)

The global block_partioner function takes an (I,J,K) sized block and partitions it the provided number of blocks

The global block_partioner function takes an (I,J,K) sized block and partitions it the provided number of blocks. The implemention is simple and can certainly be improved upon. TODO: write an optimization procedure that better load balances taking into account minimizing surface area and load balancing. The algorithm below emphasizes the minimization of surface area rather than trying to achieve perfect load balancing. This, in general, seems to be a good thing to do. In practice the load imbalance is +/- < 2 percent for the algorithm below. Even a slight increase in surface areas would seem to make things worse. But as always this needs to be verified.

17

typedef Sbint **Position**

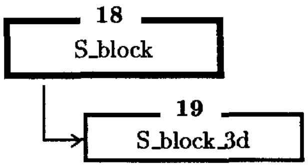
The typedef below encapsulates type of "signed index" as a position in a virtual array that may begin with a negative offset

18

class **S_block**

S_block class is a base class just here for possible future expansion of specialized 1,2,3 D structured mesh types

Inheritance

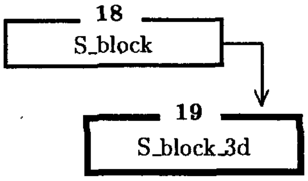


19

class **S_block_3d** : public S_block

The S_block_3d class encapsulates much of the bookkeeping involved with structured 3-D array calculations

Inheritance



Public Members

19.2 **S_block_3d** () *Default constructor* 36

S_block_3d (Position off, Index i, Index j, Index k, Index bi, Index bj, Index bk, bool os)
Constructs a block given beginning offset the number of cells in each direction, the boundary flags for each direction, and a bool flag specifing whether or not the outer surface is to be included in (i, j, k) position calculations

S_block_3d (Position off, Vector<Index>& n, Vector<Index>& bn, bool os)
Constructs a block given beginning offset, vector for cell counts, vector for the boundary flags, and a bool flag specifing whether or not the outer surface is to be included in (i, j, k) position calculations

void **initialize** (Position off, Vector<Index>& n, Vector<Index>& bn, bool os)
Converts vector (i, j, k) cell count & bool flags to index based calls

void **initialize** (Position off, Index i, Index j, Index k, Index bi, Index bj, Index bk, bool os)

Sets up a 3D block of elements for position and interior/exterior/ excluded queries

19.3 inline bool **operator()** (Index i, Index j, Index k, Position& n) 36

void **assert_bounds** (Index i, Index j, Index k)
Asserts (i, j, k) bounds are <= maximum valid indices

inline bool
excluded (Index i, Index j, Index k)
Bool test whether world coordinate (i, j, k) is to be excluded and so no number allocated in the virtual position array

inline bool
interior (Index i, Index j, Index k)
Bool test whether world coordinate (i, j, k) is in the interior of this block

inline Position&
offset () *Returns the beginning offset Position number for this block of numbers*

inline Index
size () *Return the total number of elements in this block taking into account outer surface and boundary flags*

inline Position
past_end () *Return the next available Position number after the end of this block*

inline void
cells (Index& i, Index& j, Index& k)
Returns the number of cells in (i, j, k) for this block

inline void
cells (Vector<Index>& v)
Returns the number of cells in vector v for this block

19.4 inline void **operator()** (Position n, Vector<Index>& v) 36

19.5 inline Position
operator() (Index i, Index j, Index k) 36

19.6 inline bool **operator()** (const Vector<Index>& v, Position& n) 37

inline bool
excluded (const Vector<Index>& v)
Returns true of Vector (i, j, k) is excluded from the 3-D world coordinate space

inline bool
interior (const Vector<Index>& v)
Returns true of Vector (i, j, k) is an interior entry in the 3-D world coordinate space

inline bool
interior (Position n)
Returns true of 1-D world coodinate index n is an interior entry in the 3-D world coordinate space

inline bool
exterior (const Vector<Index>& v)
Returns true of Vector (i, j, k) is an exterior entry in the 3-D world coordinate space

inline bool
exterior (Index i, Index j, Index k)
Returns true of Vector (i, j, k) is an exterior entry in the 3-D world coordinate space

Protected Members

friend class
Cfen_block *Allow Cfen_block to be a friend class*

inline Position
to_n (Index i, Index j, Index k)
Maps (i, j, k) to a single number n (i, j, k) can not be excluded based on flags set for this block

19.1 inline Vector <Index>

to_v (Position n) *Maps number n to (i, j, k) vector*
37

The S_block_3d class encapsulates much of the bookkeeping involved with structured 3-D array calculations. This class is a helper class to Cfen_block.

Manages numbers meant to be position numbers in a virtual block of Entities. Numbers can be:

Interior: Having (i,j,k) neighbors that are managed by this block (truely structured)

Exterior: On outer surface. These are required for structured indexing but they themselves do not have all their nearest neighbors being (interor/exterior).

Excluded: If the outer surface flag is not set then the outer surface is excluded from numbers managed by this block.

If the outer_surface flag is not set then the usual exterior numbers become excluded and the outer surface interior numbers become exterior.

Given a block of structured cells in 3-D space. The number of nodes, edges, faces, cells are all different. This usually leads to different counting formulas for each. Questions of whether a given node/edge/face/cell number is on the interior or exterior of a block also leads to different formulas for each situation. Things usually become tedious when finding stencils near boundarys etc... The indexing formulas are also dependent on whether or not the outer surface is to be included or not.

It is the intent of this class to provide an interface that handles mapping 3-D local coordinate (i,j,k) space to a world 1-D coordinate space and visa versa. Through the use of boundary and outer surface flags this class unifies the 1D-3D mapping formulas for nodes/edges/faces/cells indexing calculations.

Input:

offset

```
 cells in x, y, z
```

```
boundary flags for x,y z directions of type Index (not bool):
0 means excluded 1 means included
```

```
bool outer_surface flag:  true means include outer_surface
```

19.2

S_block_3d ()

Default constructor

Default constructor. The initialize member function must be called before using this object when this destructor is used.

19.3

inline bool **operator()** (Index i, Index j, Index k, Position& n)

Operator(i,j,k, &n) gives the position number n in virtual block of numbers given 3-D world coordinate (i,j,k) if interior operator() returns true else false.

19.4

inline void **operator()** (Position n, Vector<Index>& v)

Operator (Position n, Vector v) gives the world (i,j,k) coordinate number given 1-D coordinate n/

19.5

inline Position **operator()** (Index i, Index j, Index k)

Operator(i,j,k) returns 1-D world cordinate given the (i,j,k) world coordinate.

19.6

inline bool **operator()** (const Vector<Index>& v, Position& n)

Operator(Vector(i,j,k)& &n) gives the 1-D world coordinate number n given the 3-D world (i,j,k) coordinate number. Returns true if interior else false.

19.1

inline Vector <Index> **to_v** (Position n)

Maps number n to (i,j,k) vector

Maps number n to (i,j,k) vector. n must not be an excluded number.

20

class **Cfen_block**

Cfen_block class takes care of structured position bookkeeping for Cell/Face/Edge/Node (CFEN) Entities

Public Members

Cfen_block class takes care of structured position bookkeeping for Cell/Face/Edge/Node (CFEN) Entities.

Given a reference to an Reference_NC_array of Mesh_entities, and the number of cells in a (i, j, k) block, this class manages the Index positions for CFEN Entities.

The outer surface of a structured block can be included or excluded.

The Cfen_block managers numbers for 8 different Entity types of a structured block In the arrays and enumerations the following is set up

```
Array position:   enumeration:   Representation of:
-------------------------------------------------------------
   0                 no          Nodes of the block
   1                 ex           X directed edges
   2                 ey           Y directed edges
   3                 ez           Z directed edges
   4                 hx           X directed faces
   5                 hy           Y directed faces
   6                 hz           Z directed faces
   7                 ce           Cells of the block
** The 2 letter enumerations are used for compact array
   initializations where it is important to notice the
   intialization patterns.  The two letters
   {no, ex, ey, ez, hx, hy, hz, ce} are used for
   historical reasons.  Their usage is very popular in
   the FDTD community.
```

Historical note:

There are a fair number of extra member functions provided for the user. These member functions are provided for user convenience. Most of the answers could be obtained by doing simple one line adding and subtracting.

However, there could be many many more. For example, the ups()/dns() member functions could be split into dozens and dozens of combinations.

Historically the logic contained here and in S_block_3d has been distributed all over huge portions of various application codes. Applications often had many thousands lines of code devoted to such logic because of the hundreds of special case formulas usually required by these applications.

These "special case" formulas and situations are delibrately missing. The formulas have now been unified into a small set of member functions which use a series of lookup tables.

20.3

Cfen_block (R_a(Mesh_entity)& arr)

Constructs a Cfen_block given Reference_Nc_Array

Constructs a Cfen_block given Reference_Nc_Array. The user mush call initialize if this constructor is used.

20.4

inline Position **ent_type** (Position n)

Returns a number [0-7] given a position number

Returns a number [0-7] given a position number. Number returned: Element type: 0 node **1** X directed edge 2 Y directed edge 3 Z directed edge 4 X directed face 5 Y directed face 6 Z directed face 7 cell

20.5

inline bool node_excluded (Position n)

Returns boolean true if Position is an excluded node

Returns boolean true if Position is an excluded node. Note that position n is based on numbering range 0 to `of nodes in block) and not the numbering scheme used for the rest of Cfen_block.`

20.6

inline void nodes_of_cell (Position n, A_a(Index)& nodes)

Fills in the Array with 8 node positions of given cell number

Fills in the Array with 8 node positions of given cell number. Note that the node position numbers are based on included "excluded" nodes. This is NOT the numbering scheme usually used by Cfen_block. Only node_excluded() and nodes_of_cell() use this convention.

20.7

inline void up_dns (Position n, A_a(Index)& up_dn, const bool up)

Fills the ups or dns array with the positions of the nearest neighbors that are one higher or lower in dimensionality

Fills the ups or dns array with the positions of the nearest neighbors that are one higher or lower in dimensionality. This routine assumes position is in the interior so that structured position offsets are valid The boolean flag is true if ups is desired else dns

20.8

inline void **ups** (Position n, A_a(Index)& ups)

Fills the ups array with the positions of the nearest neighbors that are one higher in dimensionality

Fills the ups array with the positions of the nearest neighbors that are one higher in dimensionality. This routine assumes position is in the interior so that structured position offsets are valid

20.9

inline void **dns** (Position n, A_a(Index)& dns)

Fills the dns array with the positions of the nearest nearest neighbors that are one lower in dimensionality

Fills the dns array with the positions of the nearest nearest neighbors that are one lower in dimensionality. This routine assumes position is in the interior so that structured position offsets are valid

20.10

inline void **up_dns** (Position n, A_a(Mesh_entity*)& up_dn, const bool up)

Fills the ups or dns array with the positions of the nearest neighbors that are one higher or lower in dimensionality

Fills the ups or dns array with the positions of the nearest neighbors that are one higher or lower in dimensionality. This routine assumes position is in the interior so that structured position offsets are valid. The boolean flag is true if ups is desired else dns. The Reference_Nc_array MUST be sized to contain the Cfen block before calling this member function.

20.11

inline void **ups** (Position n, A_a(Mesh_entity*)& ups)

Fills the ups array with the Mesh_entity pointers for the nearest neighbors that are one higher in dimensionality

Fills the ups array with the Mesh_entity pointers for the nearest neighbors that are one higher in dimensionality . The Reference_Nc_array MUST be sized to contain the Cfen block before calling this member function.

20.12

inline void **dns** (Position n, A_a(Mesh_entity*)& dns)

Fills the dns array with the Mesh_entity pointers for the nearest neighbors that are one lower in dimensionality

Fills the dns array with the Mesh_entity pointers for the nearest neighbors that are one lower in dimensionality . The Reference_Nc_array MUST be sized to contain the Cfen block before calling this member function.

20.13

inline void **up_dn_address_offsets** (Position n,
A_a(SIndex)& up_dn,
const bool up)

Fills the ups or dns array with the address offsets of the nearest neighbors that are one higher or lower in dimensionality

Fills the ups or dns array with the address offsets of the nearest neighbors that are one higher or lower in dimensionality. This routine assumes position is in the interior so that structured position offsets are valid. The boolean flag is true if ups is desired else dns. The Reference_Nc_array MUST be sized to contain the Cfen block before calling this member function.

20.14

```
inline void ups_address_offsets (Position          n,
                                 A_a(SIndex)& ups)
```

Fills the ups array with the address offsets of the nearest neighbors that are one higher in dimensionality

Fills the ups array with the address offsets of the nearest neighbors that are one higher in dimensionality . The Reference_Nc_array MUST be sized to contain the Cfen block before calling this member function.

20.15

```
inline void dns_address_offsets (Position          n,
                                 A_a(SIndex)& dns)
```

Fills the dns array with the address offsets of the nearest neighbors that are one higher in dimensionality

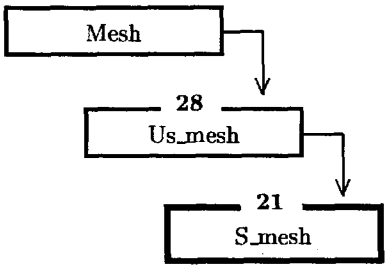
Fills the dns array with the address offsets of the nearest neighbors that are one higher in dimensionality . The Reference_Nc_array MUST be sized to contain the Cfen block before calling this member function.

21

template <class Smesh_reader> class **S_mesh** : public Us_mesh<Smesh_reader>

S_mesh class is a specialization of the Mesh class that handles the construction of a homogeneous Structured mesh part into the Entity container

Inheritance



Public Members

21.1		**S_mesh** ()	*Default constructor*	45
	Smesh_reader*	**reader** ()	*Returns a pointer to the reader being used for this mesh part*	
	virtual Mesh*	**clone** (const Sstring& mesh_file)	*Returns a pointer to an Structured Mesh part object*	
21.2	virtual void	**initialize** ()	*Initializes the Structured Mesh part*	45
21.3	virtual void	**local_build** ()	*Builds the Mesh part on this processor not communicating the outer surface to other processors*	45
	void	**close** ()	*Closes the files associated with this mesh part*	
	virtual	**~S_mesh** ()	*Default destructor*	

S_mesh class is a specialization of the Mesh class that handles the construction of a homogeneous Structured mesh part into the Entity container. The structured mesh can be cartesian or warped. A warped structured mesh has local node positions that are stored with the Node Entities. The outer surface of all Structured mesh parts are actually Unstructured. This allows structured meshes to be stitched into a single unified mesh.

21.1

S_mesh ()

Default constructor

Default constructor. Registers its name in the Mesh types static registry held by Mesh.

21.2

virtual void **initialize** ()

Initializes the Structured Mesh part

Initializes the Structured Mesh part. The nodes of the unstructured outer surfaces of this mesh file part are determined and limits of this mesh part are established during this initialization phase.

21.3

virtual void **local_build** ()

Builds the Mesh part on this processor not communicating the outer surface to other processors

Builds the Mesh part on this processor not communicating the outer surface to other processors. Both the structured and unstructured parts of this mesh part are constructed at this time.

22

#define **STD_CELLS_H_**

Std_cells

Std_cells.H

Definition of the Std_cells class.

23 class **Std_cells**

The Std_cells is a helper class for various classes Derived from the Mesh class

Public Members

23.1 **Std_cells** (int cell_type = 0)
Default constructor 48

Std_cells (const A_a(Gid)& nodes)
Constructs a Std_cell object given array of node numbers

inline void **set_cell_type** (const A_a(Gid)& n)
Determines the cell type based on the number of non-zero nodes are set in the array

inline int **cell_type** () *Returns the cell type*

inline int **n_nodes** () *Returns the number of nodes for a given cell type*

inline int **n_faces** () *Returns the number of faces for a given cell tyoe*

inline int **n_edges** () *Returns the number of edges for a given cell type*

inline Gid **n_num** (int f, int e)
Returns the local cell node number given a local face number and edge number for given face

inline Gid **e_num** (int f, int e)
Returns the local cell edge number given a local face number and edge number for given face

inline int **e_count** (int f) *Returns the number edges for a given face number*

The Std_cells is a helper class for various classes Derived from the Mesh class. Essentially, this class manages local (within a cell) numbering conventions.

The Std_cells file is temporarily "hard_wired" for hexahedral, tetrahedral, pyramid, and prism in the .C file.

(This is a minor issue. Other then input, the rest of the code should support other cell types.)

We just need to get the "element.types" like functionality in here so the user can specify any element types they want to from a file rather than editing source code. Converting tables to file input is very simple and was done in some earlier versions of TIGER. It is however on the TODO list.

TODO: move tables of Std_cells to input file.

Some of the tables can be computed from the others at reading time. (They are formulas based on previos input).

Also, high order shape elements for the moment are unsupported 2 things are needed:

```
1)Input readers which handle high-order formats for Mesh
                                          and Attributes
2)Associated length, area, area_normal etc... functions

If the topology doesn't change compared to lower
elements (which seems to be the case for most physics)
then the above mentioned items should be all that is needed
to support high order meshes.  The rest of Tiger abstracts
this complexity to the above 2 places.
```

23.1 Std_cells (int cell_type = 0)

Default constructor

Default constructor. Optional argument sets cell type for subsequent queries.

24

#define **SOLVER_H_**

Solver

Solver.H

Definition of the Solver class.

25

class **Solver**

The Solver class is the base class for all the solvers

Public Members

	Solver ()	*Default Constructor sets initialization boolean to false.*
	Solver (const Sstring& meshparts_file)	*Constructs a Solver from the given file of Mesh parts*
virtual void	**initialize** ()	*Initialize the Solver*
	~Solver ()	*Default Destructor.*

Protected Members

bool	**initialized_**	*Boolean flag signaling whether initialize() member function has been called*
static	**A_a** (Solver *)	*Holds an Array of pointers to other Solver objects that are mangaged by this Solver object*

The Solver class is the base class for all the solvers. The solver requires a pointer to a fully constructed Mesh object or a Solver input filename.

The role of this class is to manage the Solving process. Although actual solvers can be added to Derived class member functions, the intended usage most often will be to orgistrate the calling of other Solver packages such as Petsc.

```
General comment: At present, this layer of the Tiger
                 software is minimalistic.  For now we
                 are only creating a one or test cases
                 to demostrate:
                   1) Typical usage of Mesh/E_A/Utility
                      Libraries
                   2) Proof of Concept to see how much
                      bookkeeping and low level details
                      can be kept hidden from this level

Derived classes should implement the following:

Constructor: (Given Mesh*)
             During the construction phase the Solver may adorn
             the Entities contained in a Mesh with Solver specific
             Attributes.

Constructor: (Given solver_input_file)
             Reads in an existing Solver input file

Initialize():  The Solver sets up Matricies required by the
               solve() routine.

read_input_file():  Reads in an existing Solver input file

write_input_file(): Writes all required information out to
                    disk.

solve():  Invokes solver. This function is intended to be
          overloaded with various parameter arguments such as
          compute one time step or solve entire problem etc.
          The signatures well vary on the type of Solver or
          the solver package actually called.


Things to think about:
------------------------

  A given solver object should be able contain other solver objects.
     For example:
        A Finite_volume_solver may invoke:
           Absorbing_boundary_condition
           Near_to_far_zone
           Dsi_free_space
           Dsi_lossy_material
           Dsi_source
           Dsi_sensor
           ....
        each solver may want to use a common mesh
        Some solvers may want to use their own mesh
```

We may also want the output from one Solver to be the input
to another Solver.
For example:
A Near_to_far_zone solver in general may want
to use the fields a Finte_volume_solver.

Another common situation would be for a given solver to
call other member functions of the same solver object to
split the overall solving process into piecies.

This is where a high level framework could add power and
convenience.

26

class **Topology**

Names

26.1 **AT_DEC_BEGIN** (Topology, Attribute)
Topology handles the up/dn in dimensionality topology abstraction
54

void **up** (Local* loc, A_a(Mesh_entity*)& arr)
Fill in the array with the ups for the given entity

void **dn** (Local *loc, A_a(Mesh_entity*)& arr)
Fill in the array with the downs for the given entity

Mesh_entity**
up_ref (Local *loc)
return reference to the ups for given entity

Mesh_entity**
up_ref (Local *loc, Index& s)
return num ups and reference to the ups

Mesh_entity**
up_ref (Local *loc, Index& u, Index& d)
return num ups, dns and reference to the ups

Mesh_entity**
dn_ref (Local *loc)
return reference to the dns for given entity

Mesh_entity**
dn_ref (Local *loc, Index& s)
return num dns and reference to the dns

const Index&
num_ups () const *Get the number of ups these entities have.*

const Index&

num_dns () const *Get the number of downs these entities have.*

26.2 if(num_dns() > 0)(new_loc)()
(new_loc) () *Used to fix the dns starting in the last up space after a local relocate to a Species with more ups has occured* 55

void **change_up** (Local * loc, Mesh_entity *to_add, const Index pos)
Change the up at the given position to the given value.

void **change_dn** (Local * loc, Mesh_entity * to_add, const Index pos)
Change the down at the given position to the given value.

Tag the Topological properties of a mesh.

26.1

AT_DEC_BEGIN (Topology, Attribute)

Topology handles the up/dn in dimensionality topology abstraction

Topology handles the up/dn in dimensionality topology abstraction. A complexity of this abstraction is the possibility of the Entity being either structured or unstructured. If unstructured then the up/dn information is stored in local data. If up/dn information is structured then we only have stored the offsets relative to the given entity. In the structured case we add the offsets to the the given entity pounding the information into either a user provided A_a(Mesh_entity *) or our local storage space.

For methods that rely heavily on mesh topological connectivity, the Topology member functions have a huge influence on the overall application performance.

26.2

if(num_dns() > 0)(new_loc)() **(new_loc)** ()

Used to fix the dns starting in the last up space after a local relocate to a Species with more ups has occured

Used to fix the dns starting in the last up space after a local relocate to a Species with more ups has occured. This is not thread safe. The calling function should lock down the Entity.

27

#define **US_MESH_H_**

Us_mesh

Us_mesh.H

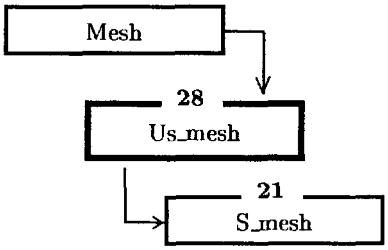
Defines the Us_mesh (Unstructured Mesh) class

28

template <class Usmesh_reader> class **Us_mesh** : public Mesh

Us_mesh class is a specialization of the Mesh class that handles the construction of a volumetric homogeneous Unstructured mesh part into the Entity container

Inheritance



Public Members

28.4 **Us_mesh** () *Default constructor* 60

inline Usmesh_reader*
reader () *Returns a pointer to the reader being used for this mesh part*

void **open** (const Sstring& file_name, const Sstring mess)
Opens the files associated with the the provided filename and initializes the reader

virtual Mesh*
clone (const Sstring& mesh_file)
Returns a pointer to an Unstructured Mesh part object

28.5 virtual void
initialize () *Initializes the Unstructured Mesh part* 61

virtual void

local_build () *Builds the Mesh part on this processor not communicating the outer surface to other processors*

28.6 Mesh_entity*
build_cell (int, A_a(Mesh_entity*)&, Species*)
Builds all the Entities required to add a cell into the Entity container 61

inline Mesh_entity*
find_common_up (Mesh_entity *a, Mesh_entity *b)
Finds a common up entity given two entities

inline Mesh_entity*
create_edge (Mesh_entity *n1, Mesh_entity *n2)
Creates an edge given two node entities

inline Mesh_entity*
create_face (A_a(Mesh_entity *)& cells_edges,
int *edges, int num_edges)
Creates a face given set of edge entities

inline Mesh_entity*
create_cell (A_a(Mesh_entity *)& faces,
int num_faces, Species *cell_species)
Creates a cell given a set of face entities

Sstring **part_n_reader_name** ()
Returns a Sstring concatenating the Mesh part type name and the Mesh_reader type names

Sstring **mesh_part_file** () *Returns the mesh part file used to construct this Us_mesh object*

void **close** () *Closes the files associated with this mesh part*

virtual **~Us_mesh** () *Default destructor*

Protected Members

28.1 Std_cells **connect** *Holds local cell connectivity* 62

int	**nodes_built**	*Holds the number of nodes constructed and inserted into Mesh's Entity container*
int	**edges_built**	*Holds the number of edges constructed and inserted into Mesh's Entity container*
int	**faces_built**	*Holds the number of faces constructed and inserted into Mesh's Entity container*
int	**cells_built**	*Holds the number of cells constructed and inserted into Mesh's Entity container*
Usmesh_reader	**reader_**	*Holds the unstructured Mesh_reader object*
	A_a (S_a(Attribute *))	*Holds an array of Attributes lists associated with common edge Mesh_entity currently being constructed*
	A_a (Species*)	*Holds an array of pointers to possible Species' for the edge currently being constructed*
	A_a (S_a(Attribute *))	*Holds an array of Attributes lists associated with common face Mesh_entity currently being constructed*
	A_a (Species*)	*Holds an array of pointers to possible Species' for the face currently being constructed*
	A_a (Gid)	*An Array of the original Node_id's read in from the Mesh input file*
	A_a (Vector<Mesh_position_type>)	*An Array of the original node positions read in from the Mesh input file*
static int	**reference_count**	*Holds a reference count of how many Us_mesh parts have been instanciated to date*

28.2 **Us_mesh** (const Sstring& cls_name)
Constructor that registers the given class name in the Mesh types static registry held by Mesh 62

28.3 Sstring **part_n_reader_name** (const Sstring& cls_name)
Returns a Sstring concatenating the given Mesh part type name and the Mesh_reader type names 62

inline void **add_att** (const Sstring& att_name,
S_a(Attribute *)& att)
Given a Sstring this member function finds and inserts the Attribute into the provided Sorted array

void **edge_face_att_species_setup** ()
Sets up the Edge and Face Attribute and Species arrays used in the Unstrctured grid construction process

Us_mesh class is a specialization of the Mesh class that handles the construction of a volumetric homogeneous Unstructured mesh part into the Entity container. The Us_mesh can add n-faced - m-sided elements into a Mesh. However, the class is templated based on a Mesh_reader type that must know how to read these arbitrary cell formats. Furthermore, the class aggregates a Std_cells class object that must know about the local connectivity of a given cell type. So in order to support a new type of cell the user must add to the Std_cells class and a provide or enhance a reader.

28.4

Us_mesh ()

Default constructor

Default constructor. Registers its name in the Mesh types static registry held by Mesh.

28.5

virtual void **initialize** ()

Initializes the Unstructured Mesh part

Initializes the Unstructured Mesh part. The nodes of the unstructured mesh file part are read in during this initialization phase. Also, limits of the mesh part are determined.

28.6

Mesh_entity* **build_cell** (int, A_a(Mesh_entity*)&, Species*)

Builds all the Entities required to add a cell into the Entity container

Builds all the Entities required to add a cell into the Entity container. Given a set of node ids this member function loops over every face of the cell and every edge of every face creating any Mesh entities that are required. The following is a simple outline of the procedure.

```
Convert node numbers to Entities
For every face
   For every edge
      If it doesn't exist
         create edge
      else
         find edge
   If face does not exist
      create face
   else
      find_face
given faces
create cell
```

28.1

Std_cells **connect**

Holds local cell connectivity

Holds local cell connectivity. "Local" here denotes node, edge, and face indexing relative to a single cell.

28.2

Us_mesh (const Sstring& cls_name)

Constructor that registers the given class name in the Mesh types static registry held by Mesh

Constructor that registers the given class name in the Mesh types static registry held by Mesh. Only Derived classes can access this member function.

28.3

Sstring **part_n_reader_name** (const Sstring& cls_name)

Returns a Sstring concatenating the given Mesh part type name and the Mesh_reader type names

Returns a Sstring concatenating the given Mesh part type name and the Mesh_reader type names. This member function can only be called by Derived class member functions.

29 class **Tiger**

The Tiger Class manages the construction and application of Mesh and Solver objects at a high level

Public Members

	void	**initialize** (int argc, char** argv)	*Loads the Attribute data base and Attribute map files if this is the first time this member function is called*	
29.1		**Tiger** (int argc, char** argv)	*Default constructor*	64
29.2		**Tiger** (int argc, char** argv, const Sstring& filename)	*Loads the Attribute data base and Attribute map files*	64
	void	**mesh** (const Sstring& filename)	*Constructs a Mesh consisting of the Mesh parts contained in filename*	
	Mesh*	**mesh** ()	*Returns a pointer to the currently constructed mesh*	
29.3	void	**geid_sync** ()	*Syncs the Entities by providing a unique global Entity identification number (geid) accross all processors*	65
29.4	inline bool	**geid_synced** ()	*Returns boolean answering the question of whether the geid_sync() member function has been called*	65
	void	**load_sources** (const Sstring& sources_filenames)	*Reads in Sources input relavent to the current Mesh and Solver*	
	void	**load_sensors** (const Sstring& sensors_filenames)		

The Tiger Class manages the construction and application of Mesh and Solver objects at a high level. Various stages of the preprocessor can be called from this class. A typical usage would be calls from a gui or script file drivers.

29.1

Tiger (int argc, char** argv)

Default constructor

Default constructor. Loads in the Attribute data base and Attribute map files. If this is the first Tiger object.

29.2

Tiger (int argc, char** argv, const Sstring& filename)

Loads the Attribute data base and Attribute map files

Loads the Attribute data base and Attribute map files. Then constructs a Mesh consisting of the Mesh parts contained in filename.

29.3

void **geid_sync** ()

Syncs the Entities by providing a unique global Entity identification number (geid) accross all processors

Syncs the Entities by providing a unique global Entity identification number (geid) accross all processors. In general, geid's will not be contiguous. After calling this member function no totally new Entities can be created. New Entities created on a given processor must have existed on some other processor before calling this geid_sync function.

29.4

inline bool **geid_synced** ()

Returns boolean answering the question of whether the geid_sync() member function has been called

Returns boolean answering the question of whether the geid_sync() member function has been called. If geid_synced() == true then use of geids valid.

29.5

~Tiger ()

Default destructor

Default destructor. Deletes the mesh if one exists.

Class Graph